ASSEMBLER-PRAXIS

AUF ATARI ST

ATARI 260ST, ATARI 520ST, ATARI 1040ST

- Erklärt Grundlagen:
  - fundierte Begriffserklärungen zu Assemblern und Programmstrukturen
  - systemkundige Funktionserklärung der 68000-Assemblerbefehle
- Zeigt Anwendungen:
  - gut kommentierte Musterprogramme
  - gut kommentierte Systemanwendungen
- Entwickelt Utilities:
  - kommentierte BIOS/GEM-DOS-Funktionen als Programmierungshilfen
  - Werkzeugkasten für TOS-Anbindungen u. ä.

Roland Löhr

tewi

Inhaltsverzeichnis

Seite

Vorwort VII

Einleitung 1
1. Merkmale der CPU 68000 3
2. Die Exceptions 9
3. Assembler und Syntax 15
4. Vom Text zum Programm 21
5. Leistungen eines Textprogrammes 27
6. Die typische Befehlszeile 29

Das Sprachreservoir 39
7. Die Befehlsgruppen 41
8. Datenübertragungen 47
9. Spezialbefehle für Adreßregister 51
10. Statusbefehle 55
11. Status prüfen und merken 59
12. Binäre Rechenarten 61
13. Dezimale Rechenarten 65
14. Logische Befehle 67
15. Vergleiche 69
16. Befehle zur Bitprüfung 73
17. Schiebe- und Rotierbefehle 75
18. Sprünge, Unterprogramme, numerierte Traps 77
19. Bedingte Verzweigungen, Schleifenkonstrukte 81
20. Stackverwaltung, verschiedene Befehle 85
21. Adressierungsarten – die effektive Adresse 91
22. Die wichtigsten Direktiven 99
23. Sonstige Direktiven 105

Angewandte Assembler-Programmierung 111
24. Das Betriebssystem 113
25. Ein erstes Programm 117
26. Die BIOS-Funktionen 127
27. Eine BIOS-Toolbox 133
28. Ein GEMDOS-Toolkit 139

29. Das erweiterte XBIOS 147
30. Benutzung des Toolkits 151

Mit Texten arbeiten 155
31. Fallentscheidungen 157
32. Filter 161
33. Ein Editor 165
34. Ein Parser 169
35. Strings suchen 175

Spezielle Ein- und Ausgaben 181
36. Tastatur-Dekodierung 183
37. Memory-Dump 187
38. Floppy-Test 193
39. Floppy-Funktionen 197
40. Mit Dateien arbeiten 205
41. Serielle Datenübertragung 215

Rechenroutinen und Datenfelder 221
42. Binäres Rechnen 223
43. Dezimales Rechnen 233
44. Zahlenwandlung 237
45. Arrays 243

Nachwort 251

Anhang 253

Literaturhinweis 257

Vorwort

Wer sich schon etwas mit der maschinensprachlichen Programmierung eines Computers befaßt hat, kennt die üblichen Startschwierigkeiten: Man muß zunächst lernen, das Gerät richtig zu bedienen. In der nächsten Stufe befaßt man sich mit der Hantierung des Textprogrammes (Editor) und des Assemblers. Sodann muß man sich mit dem Befehlssatz der Zentraleinheit und vielleicht auch der Peripheriebausteine befassen. Erst nach diesen Vorbereitungen kann man an die Programmierung gehen. Wenn man dabei auf gut erklärte Programmteile und Programme zurückgreifen kann, dann wird die eigene Einarbeitung erleichtert.

Dieses Buch geht von geringen Grundkenntnissen in der Assembler-Programmierung aus. Es gibt zunächst einmal viele Hinweise zu den sprachlichen Regeln und zur Hantierung des Assemblers. Wer einen Computer programmiert, muß auch etwas von seiner Hardware verstehen. Die notwendigen Erläuterungen dazu sind so kurz wie möglich gehalten. Sie beschränken sich im wesentlichen auf die Register der Zentraleinheit 68000. Einen größeren Raum nimmt die Darstellung des Befehlssatzes ein, der zu Funktionsgruppen zusammengefaßt wird, um die Übersicht auch beim Nachschlagen zu verbessern.

In der eigenen Programmierung von Atari-Rechnern benötigen wir immer wieder die Dienstleistungen ihres Betriebssystems TOS. Dieses Buch ist daher sehr auf die Erklärung seiner Grundfunktionen in den Ebenen BIOS, XBIOS und GEMDOS abgestellt. Nach und nach wird ein Werkzeugkasten, d. h. eine kleine Quelltext-Bibliothek, für die Benutzung der wichtigsten Funktionen des Betriebssystems eingerichtet. Mit ihnen werden im Hauptteil beispielhaft höhere Dienstleistungen entwickelt, die man im Programmieralltag bei immer wiederkehrenden Aufgabenstellungen weiterbenutzen kann. Diese Beispiele sollen vor allem aber den leistungsfähigen Befehlssatz der CPU 68000 in möglichst vielen Varianten und Adressierungsarten lehrend darstellen.

Behandelte Themen sind dabei u. a.: Texteingabe (Editor), Texterkennung, Fallentscheidungen, Arbeiten mit der Floppy, serieller Rechnerverbund, binäres und dezimales Rechnen und das Arbeiten mit Arrays (Datenbank).

Dem te-wi Verlag, besonders Herrn Michael Mackel, ist für die Anregung zu danken, daß den Betreibern eines Atari-ST ein Buch mit viel Hantierungshinweisen und Software vorgelegt werden sollte. Der Autor hofft, daß es nicht nur ihnen, sondern auch Betreibern anderer 68000-Systeme nützliche Anregungen für die eigentlich noch immer schöne Programmierung in Assembler geben möge. – Besonderer Dank gilt meiner Frau Renate für ihre Geduld und ihr Wirken während der Niederschrift und der oft langwierigen Erprobung der Programme.

Ahrensburg, im Juni 1986

Roland Löhr

EINLEITUNG

EINLEITUNG

Kapitel 1

Merkmale der CPU 68000

Wenn wir den Atari 520ST programmieren, dann müssen wir auch die Hauptmerkmale seiner Zentraleinheit MC68000 von Motorola kennen.

Zur Hardware

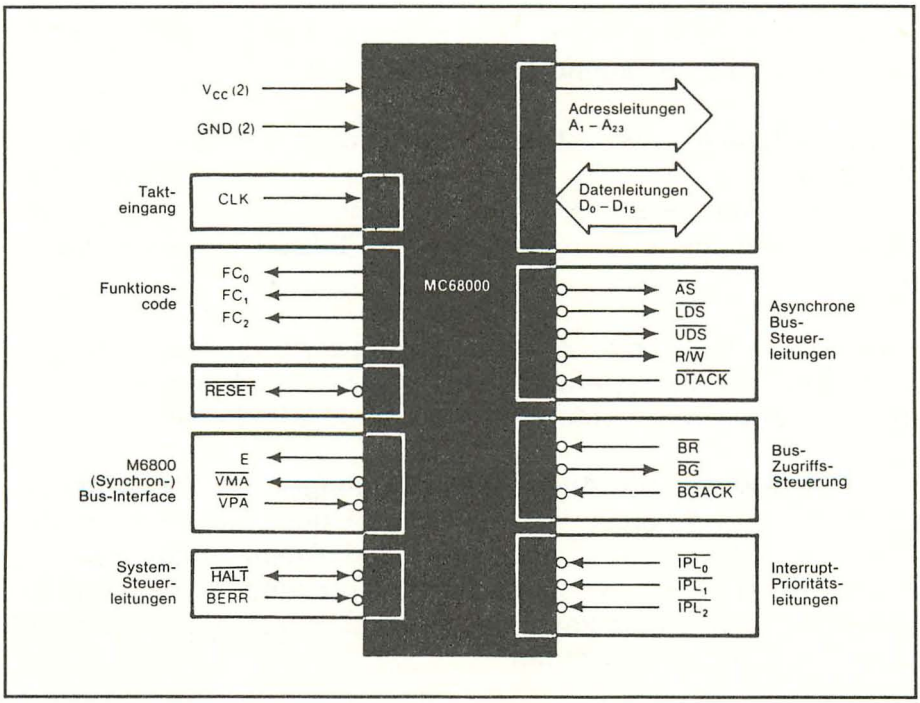


Abbildung 1: Einteilung der Leitungen des M68000

Die CPU 68000 verwirklicht ein modernes und sehr leistungsfähiges Design:

Durchgehender Adreßraum von 16 MB, der über 24 Adreßbuspins erreicht wird, und zwar ohne Multiplexing der Signale und ohne die Notwendigkeit, intern Bankregister führen zu müssen.

Datenbus mit 16 Leitungen für hohen Durchsatz, auch hier kein Multiplexing.

Hauptregister mit 32 Bits Breite. Gleichwohl können die meisten Befehle auf eine Arbeitsbreite von 8 Bit (Byte), 16 Bit (Wort) oder 32 Bit (Langwort) eingestellt werden.

Es gibt keine besonderen Lese- und Schreibbefehle für die Peripherie. Wie auch in den Prozessorfamilien 68xx und 65xx werden Interfacebausteine wie normale Speicherzellen angesprochen, sie sind **"memory mapped"**.

Befehlslängen, die entweder nur 1 Wort umfassen oder ein Vielfaches von 16 Bit. Es gibt also keine Befehle, die eine ungeradzahlige Menge Bytes haben. Viele Befehle können in Adressierungsarten benutzt werden, die nur 1 Wort je Gesamtbefehl erfordern.

Zugriffe auf Befehle, Worte und Langworte müssen immer an geradzahligen Speicheradressen erfolgen, andernfalls erzeugt die CPU eine Programmausnahme wegen Adreßfehlers.

Klares System für Ausnahmebedingungen im Programmablauf (Exceptions, auch Interrupte), das über Vektoren in den niedrigen Adressen zu einer gezielten Ausnahmebearbeitung mit bekannter Ursache führt. Bei entsprechender Beschaltung und Programmierung kann der Prozessor damit praktisch absturzsicher betrieben werden.

Interrupte können sich in 7 Prioritätsebenen an den Pins der CPU bemerkbar machen. Interrupte niedriger Priorität können mit der Interruptmaske im Status am Vordringen gehindert werden.

CPU ist eine Handshake-Maschine, die asynchron mit dem Speicher und mit der Peripherie zusammenarbeitet: Vor dem Schreiben und nach dem Lesen wartet sie auf das Signal DTACK, mit dem die angesprochene Einheit sich als existent und arbeitsbereit melden muß. Eine Überwacherschaltung (Watchdog) im Computeraufbau achtet darauf, daß die langsamste Einheit sich in einer gewissen Toleranz melden kann. Trifft das DTACK nicht rechtzeitig ein, so erzeugt sie das Signal BERR (Busfehler), das zu einer Ausnahmeverarbeitung führt.

Die übrigen elektrischen Signale sind für den Programmierer nicht so wichtig, sie gehören zum System-Design. Erwähnenswert sind die drei Pins FC0 ... FC2 für die Ausgabe des Funktions-Codes. Hier kann man ablesen, welche Art von Zyklus die Zentraleinheit gerade ausführt. – Die CPU kann ferner veranlaßt werden, die Busse freizugeben und anzuhalten, damit eine externe Einheit (z. B. eine Festplatte) per DMA (direkter Speicherzugriff) Informationen aus dem Computer holen oder in seinen Speicher bringen kann. Für die Anforderung der Busse und ihre Freigabe gibt es ein geregeltes Protokoll.

Daten- und Adreßregister

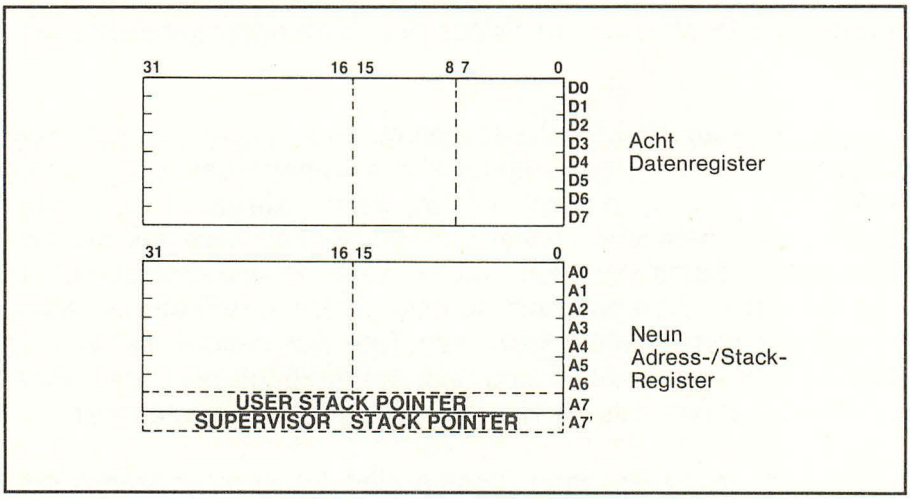


Abbildung 2: Daten/Adressregister

Die für den Anwender erreichbaren Register sind die 8 Datenregister D0 ... D7, die 8 Adreßregister A0 ... A7 und der Programmzähler. Sie sind jeweils 32 Bit breit (Numerierung 0 ... 31). Daneben gibt es das Statusregister (SR) in Wortbreite mit dem System-Byte und dem Bedingungscode-Register, auch User-Byte.

In der Registerzeichnung sind zwei Adreßregister A7 und A7' enthalten, bezeichnet als Anwender-Stackpointer und Supervisor-Stackpointer. Es sind tatsächlich elektrisch unabhängige Register in der CPU, die als Hardware-Stackpointer dienen, wenn Unterprogrammaufrufe zur Ablage der Rückkehradresse führen. Welches gerade aktiv ist, hängt vom Supervisor-Bit 'S' im Status ab. Für S = 0 ist der Anwender-Stackpointer aktiv, für S = 1 des Systemstackpointer. Für den mit S = 1 gegebenen Supervisor-Modus gelten besondere Bedingungen (s. u.).

Das Statusregister

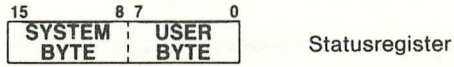


Abbildung 3: Statusregister

Das obere Byte des Statusregisters (Wort) ist das System-Byte. Hier sind 5 Bits benutzt: der Einzelschritt-Modus (T = 1); der Supervisor-Modus (S = 1); die Interrupt-Maske (3 Bit). Sie steht in Korrespondenz zu den Eingabepins IPL0 ... IPL2 der CPU. Ein elektrisch sich mit binär '111' meldender Interrupt hat die höchste Priorität, es ist quasi ein NMI (Non Maskable Interrupt), der immer bearbeitet wird. Ein mit '001' sich meldender Interrupt ist von geringstem Vorrang. Ein an IPL0 ... IPL2 ankommender Interrupt muß einen größeren Wert haben, als die Interruptmaske im Statusregister, damit er zur Bearbeitung kommt.

Für den Einzelschritt-Modus (T = 1) gibt es in den niedrigen Adressen einen eigenen Ausnahmevektor, mit dessen Programm der Systemprogrammierer für Anzeigen an den Benutzer sorgt.

Nur im Supervisor-Modus (S = 1 im Status) können die sog. privilegierten Befehle ausgeführt werden, die im Anwender-Modus (S = 0) zu einer Ausnahme wegen Privilegverletzungen führen (eigener Ausnahmevektor). Es sind dieses besonders Befehle, die das System-Byte verändern sollen. Ein Anwenderprogramm soll also nicht in der Lage sein, das Betriebssystem umzuwerfen. Gleichwohl muß man sich vor Augen halten, daß alle Programmausnahmen (Exceptions) die CPU in den Supervisor-Modus versetzen. Damit sind hier Manipulationsmöglichkeiten gegeben.

Exceptions legen die Rückkehradresse des unterbrochenen Programmteils auf dem Supervisor-Stack ab, auch wenn vorher der Anwender-Modus herrschte.

Das Bedingungscode-Register

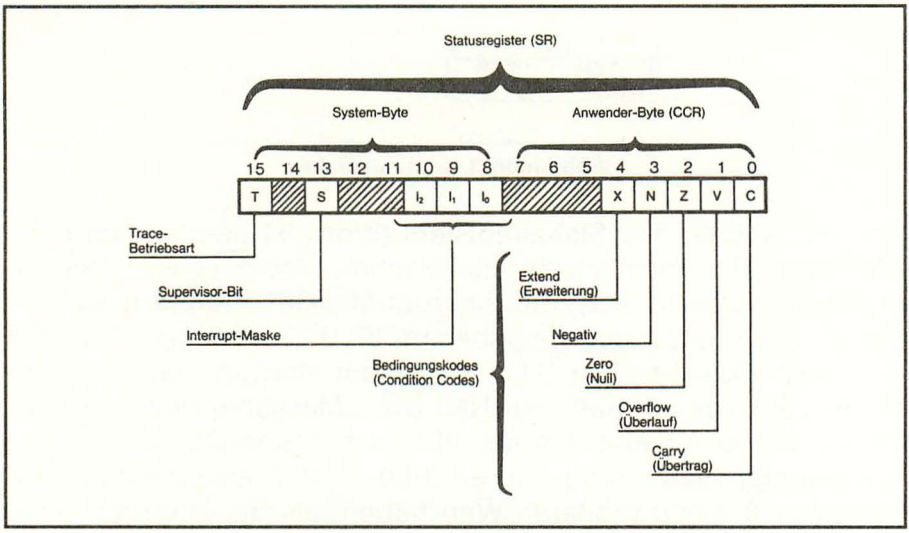


Abbildung 4: Statusregister-Bits

Das niederwertige Byte des Statusregisters ist das Bedingungscode-Register (CCR), das die Ergebnisse der Befehle widerspiegelt, soweit sie (meist) Bedingungscodes verändern. Hier sind 5 Bits benutzt, an denen man im wesentlichen ablesen kann:

X = Extendbit bei Addition, Subtraktion und Verschiebungen
N = Anzeige, ob negativ oder positiv
Z = Anzeige, ob Null oder ungleich Null
V = Überlauf bei arithmetischen Operationen
C = Übertrag aus arithmetischen Operationen, Vergleichen und Verschiebungen.

Die Exceptions

Es gibt viele Bedingungen, die ein Programm unterbrechen können. Zu ihnen gehören nicht nur Interruptanforderungen von außen und Unterbrechungen durch Programmbefehle (Software-Interrupts mit TRAP #), sondern auch Systemfehler, Division durch Null, Benutzung nicht implementierter Befehlscodes usw.

Ausnahmevektoren

Die Adressen 000000 ... 0003FF eines voll initialisierten Computers sind für 256 Langwort-Vektoren vorgesehen. Mit den Adressen 000000 ... 000007 spricht man beim kalten RESET normalerweise ein PROM an. Sein erstes Langwort ist der Supervisor-Stackpointer, den die CPU zuerst nach A7' lädt, das zweite ist der Programmzähler, mit dem das System beginnt. Dahinter können sich im RAM bis zu 254 Vektoren anschließen, für die eine Tabelle mit Erklärungen folgt.

Adresse Hexa-dezimal	Bedeutung des Vektors	Vektor-nummer Hexa-dezimal
000	Anfangsadresse Supervisor-Stackpointer (bei RESET)	00
004	Anfangsadresse Programmzähler (bei RESET)	01
008	Busfehler	02
00C	Adressfehler	03
010	nicht implementierter Befehl	04
014	Division durch Null	05
018	Befehl CHK	06
01C	Befehl TRAPV	07
020	Privilegverletzung	08
024	Trace	09
028	Befehlscode 1010 Emulator	0A
02C	Befehlscode 1111 Emulator	0B
030	reserviert	0C
034	reserviert	0D
038	reserviert	0E
03C	nicht initialisierter Interrupt	0F
04C 05C	reserviert	10 – 17[3])
060	falscher Interrupt	18
064	Interrupt-Autovektor (Ebene 1)	19
068	Interrupt-Autovektor (Ebene 2)	1A
06C	Interrupt-Autovektor (Ebene 3)	1B
070	Interrupt-Autovektor (Ebene 4)	1C
074	Interrupt-Autovektor (Ebene 5)	1D
078	Interrupt-Autovektor (Ebene 6)	1E
07C	Interrupt-Autovektor (Ebene 7)	1F
080 0BC	Trap-Befehlsvektoren #0 bis #15	20 – 2F
0C0 0FC	reserviert	30 – 3F
100 3FC	Anwender-Interruptvektoren (Nonautovektoren)	40 – FF

Abbildung 5: Ausnahmevektoren

Wichtig für die Programmierung des Atari 520ST und für Systeme mit dem Betriebssystem CPM-68K von DRI sind vor allem 16 Trap-Vektoren im Adreßraum $80-$BF für die Befehle TRAP #0 ... TRAP #15 für Software-Interrupte.

TRAP-Befehle	Vektoradresse:	Befehl:	Vektornummer:
	$80 – $83	TRAP #0	$20
	$84 – $87	TRAP #1	$21
	$88 – $8B	TRAP #2	$22
	$8C – $8F	TRAP #3	$23
	$90 – $93	TRAP #4	$24
	$94 – $97	TRAP #5	$25
	$98 – $9B	TRAP #6	$26
	$9C – $9F	TRAP #7	$27
	$A0 – $A3	TRAP #8	$28
	$A4 – $A7	TRAP #9	$29
	$A8 – $AB	TRAP #10	$2A
	$AC – $AF	TRAP #11	$2B
	$B0 – $B3	TRAP #12	$2C
	$B4 – $B7	TRAP #13	$2D
	$B8 – $BB	TRAP #14	$2E
	$BC – $BF	TRAP #15	$2F

Abbildung 6: TRAP-Befehle

Weitere Ausnahmebedingungen

Ein Busfehler wird erzeugt, wenn kein DTACK-Signal als Handshake eintrifft. Ein Adreßfehler entsteht bei Ansprache einer ungeraden Adresse während des Zugriffes auf Befehle, Worte und Langworte. Die Begriffe illegaler Befehl, Division durch Null und Privilegverletzung in der Tabelle sprechen für sich. Der CHK-Vektor (Nr. $06 in der Tabelle) wird benutzt, wenn ein Datenregister eine obere oder untere Grenze über-/unterschreitet.

Beim 68000 sind Befehlscodes mit $Axxx und $Fxxx nicht implementiert, sie kommen erst beim 68020 zum Einsatz. Solche Befehle führen zu einer Programm-Ausnahme. Über die entspre-

chenden Vektoren können Routinen angesprochen werden, die Befehle des 68020 mit anderen Mitteln nachbilden. Der Atari 520ST tut das nicht, verwendet jedoch $Axxx für "A-Line"-Befehle.

Interruptvektoren 25 . . . 31

Die Vektoren 25 ... 31 werden benutzt, wenn Interruptsignale an den drei Interruptpins der CPU eintreffen. Das binäre Muster der Signalkombination wird dabei zugleich als Prioritätsstufe betrachtet. – Wenn eine Interruptquelle nach einer gewissen Wartezeit kein Handshake abgibt, kann auf unechten Interrupt erkannt werden (Vektor $18). Das kann im Zusammenhang z. B. mit zufälligen Induktionsfunken stehen.

Autovektor-Interrupts

Vektoradresse:	Prioritätsebene:	Vektornummer:	Priorität:
$64 – $67	1 (niedr. Pr.)	$19	niedrigste
$68 – $6B	2	$1A	↓
$6C – $6F	3	$1B	
$70 – $73	4	$1C	
$74 – $77	5	$1D	
$78 – $7B	6	$1E	
$7C – $7F	7 (NMI)	$1F	höchste

Abbildung 7: Auto-Vektoren

Die weitere Verfolgung eines Interrupts hängt von der Beschaltung der Interruptquelle und von seiner Fähigkeit ab, ggfs. eine Vektor-Nummer auf den niedrigen Datenbus abzugeben. Antwortet die Interruptquelle mit dem Signal VPA, aber ohne DTACK, so wird entsprechend ihrer Priorität einer der Vektoren $19 ... $1F genommen (= Auto-Vektor). Antwortet sie mit DTACK, so muß sie eine Nummer auf den niedrigen Datenbus ausgeben. Normalerweise liegt sie zwischen dezimal 64 und 252. Der Programmzähler wird dann entsprechend aus dem

RAM-Bereich ab Adresse $000100 geladen. – Diese zweibahnige Lösung ermöglicht es, sowohl mit Interfacebausteinen der Familie 68xx umzugehen, als auch mit modernen der Familie 68xxx, die auch schon intern die Nummer ihrer Interruptabgabe bei mehreren Quellen zuordnen können. In der Summe: Der MC68000 und seine Verwandten kommen ohne umständliches Feststellen der Interrupt-Quelle aus (Interrupt Polling). – In jedem Einzelfall studiere man die Datenblätter.

Kapitel 3

Assembler und Syntax

Assembler-Sprachen sind von Systemprogrammierern geschaffene Sprachen, in denen der Computeranwender mit Befehlen und Anweisungen seine Problemlösung als 'Quelltext' niederschreiben kann. Mit dem Wort **Assembler** bezeichnet man dagegen ein Übersetzungsprogramm, das aus solchen Quelltexten ein 'Maschinenprogramm' erzeugt, das anschließend im Computer ablaufen kann. Da Computer nur elektrische Ja/Nein-Informationen verarbeiten können, ist ein Maschinenprogramm daher letztlich ein Strom binärer Informationen.
Elemente von Assemblersprachen sind symbolische Namen und Labels, Zahlen verschiedener Zahlensysteme, arithmetische und logische Operatoren sowie Sonderzeichen. Eine Befehlszeile in 68000-Assemblern wird typisch wie folgt geschrieben:

label **BEFEHL.**operationsbreite quelle,ziel kommentar

Assembler-Sprachen und Assembler bedingen einander: Beim Schreiben eines Assemblers hat der Systemprogrammierer eine Vorstellung davon, wie der Eingabestrom eines Quelltextes aussehen sollte, damit er vom Assembler in den binären Ausgabestrom des Maschinenprogramms umgewandelt werden kann. Sobald der Assembler fertig ist, liegen auch die in der dazugehörigen Assembler-Sprache zulässigen Vokabeln und sprachlichen Ausdrucksweisen fest, deren Regeln man Assembler-Syntax nennt. Es hängt also vom Systemprogrammierer ab, nach

welcher Grammatik Sätze in der Assembler-Sprache seines Assemblers formuliert sein dürfen.

Für die CPU MC68000 gibt es tatsächlich verschiedene Assembler und Assembler-Sprachen. Bei der Darstellung der Syntax, des Sprachgebrauches und der Hantierung des Assemblers auf Atari 520ST-Computern gehen wir auf den Assembler ein, der von Atari mit dem Entwicklungspaket für Systemprogrammierer geliefert wird. Dieser Assembler wurde von der amerikanischen Firma Digital Research (DRI) geschrieben. Der von ihm geforderte sprachliche Ausdruck ist ziemlich deckungsgleich mit dem Assembler von Motorola für die CPU MC68000. Der DRI-Assembler kommt auch auf anderen Computern zum Einsatz, die unter dem Betriebssystem CP/M-68K von DRI arbeiten. – Der Anwender mag sich aber auch z. B. für den gut dokumentierten Assembler der englischen Firma GST entscheiden, der zusammen mit einem geeigneten Textsystem geliefert wird. Auch dieser Assembler folgt der Motorola-Syntax.

Wenn wir im weiteren Text vereinfachend nur vom "Assembler" sprechen, so geschieht das wegen der gegenseitigen Entsprechung von Assembler-Sprache und dem dazu gehörenden Übersetzungsprogramm.

Grundelemente einer Assembler-Sprache sind zunächst Vokabeln und Trennzeichen zwischen ihnen. Bei Vokabeln denken wir in erster Linie an Befehle oder 'Opcodes' für die CPU wie z. B. 'Branch to Subroutine' (Verzweige zu einem Unterprogramm). Wenn man auch Assembler entwickeln könnte, die solche längeren Befehle in Deutsch oder Englisch verstehen oder verwerten, so hat es sich schon wegen der geringeren Arbeit beim Eintippen von Programmen als zweckmäßig erwiesen, mit mnemonischen Befehlsabkürzungen zu arbeiten, in vorstehenden Fall also mit 'BSR'. (Motorola nennt diese Befehlsabkürzungen auch mnemonic machine instruction operation codes.)

Zum Sprachumfang gehören auch mnemonische Abkürzungen für Register der Zentraleinheit, so z. B. A5 oder D0 für Adreßregister 5, Datenregister 0.

Vokabeln sind aber auch Anweisungen an den Assembler selbst, so z. B. "lege ab erreichter Stelle des Zuordnungszählers eine Konstante ab". Eine solche Anweisung steuert wie ein Befehl den Assemblerlauf, nicht aber das erzeugte Maschinenprogramm, und wird daher auch Pseudo-Opcode genannt. Es gibt eine Reihe weiterer Assembleranweisungen oder 'Direktiven', die zu den Vokabeln der Assemblersprache gehören, aber nur Wirkung auf den Assembler haben, so z. B. .end. Diese Vokabel bedeutet: Hier ist das Ende des Quelltextes. Pseudo-Opcodes und Direktiven beginnen meistens mit einem Punkt.

In gesprochenen Sprachen und ihren Schriften gibt es Trennzeichen wie die Sprechpause oder das Zwischenraumzeichen auf dem Papier. Soweit keine Silbentrennung erfolgt, ist auch ein Zeilenende ein Trennzeichen. Nicht anders ist es in den meisten Assemblern: Zwischenräume (englisch: white space) und das/die Zeilenende-Zeichen sind Trenner. (In anderen Sprachen als der hier betrachteten werden auch andere Trenner benutzt.) In unserem Assembler ist ferner das Komma ein wichtiger Trenner: Es trennt die Bezeichnung von Operanden innerhalb eines Befehles, z. B. D0 von D5 in MOVE.L D0,D5. Weil der MC68000 eine Zweiadreß-Maschine ist, bedeutet das: Kopiere den Inhalt des Datenregisters D0 in das Datenregister D5.

Auch der Punkt kann ein Trenner sein, wie wir soeben bei MOVE.L D0,D5 gesehen haben. Die mnemonische Abkürzung des Befehles reicht bis einschließlich 'E' in MOVE. Das '.L' steht für die Abwandlung des Befehles auf die Spielart 'Übertrage ein Langwort'. Andere Spielarten wären hier .W (Wortbreite) und .B (Byte).

Elemente der Assemblersprache sind ferner die Ziffern 0 ... 9, die hexadezimalen Ziffern A ... F in hexadezimalen Ausdrücken sowie Sonderzeichen wie $, *, ; und Operatoren wie + und –. Wir kommen im einzelnen darauf zurück.

Eine der großen Bequemlichkeiten, die Assemblersprachen dem Maschinenprogrammierer bieten, liegt in der möglichen Verwendung von Symbolen: Er kann z. B. Programmstellen, die als Verzweigungs- oder Sprungziel dienen sollen, symbolisch mit einem Namen oder 'Label' versehen. Er braucht sich dann nicht mehr darum zu kümmern, unter welcher tatsächlichen Adresse dieses Label später auf dem Computer angesprochen werden muß. Zu den Symbolen gehören ferner die Definitionen, die man dem Assembler typisch in den ersten Zeilen eines Moduls mitteilt, z. B. LINEFEED .EQU $0A. Das Symbol LINEFEED soll den Wert hex 0A bzw. dezimal 10 haben. – Vom Anwender geschaffene Symbole gehören nicht zu den Vokabeln des Assemblers. Die Art ihrer Benutzung ist jedoch in der Bildungsregel für Programmzeilen (Syntax) vorgeschrieben.

Der DRI-Assembler arbeitet zeilenorientiert: Ein Befehl oder eine Anweisung muß vollständig in einer Zeile enthalten sein, und je Zeile kann nur ein Befehl geschrieben werden. Das Zeilenende-Zeichen (CR) ist der Trenner für Sätze der Assemblersprache. (Es gibt Assembler, die mehrere Befehle je Zeile zulassen, wenn bestimmte Trennzeichen benutzt werden.) Innerhalb einer Programmzeile muß man ferner eine gewisse Reihenfolge für Label, Befehle und Operanden beachten. Man spricht vom 'Format einer Eingabezeile'. Die Formatierungsregeln werden in einem besonderen Abschnitt besprochen. Hier eine typische Befehlszeile als Muster:

MAIN MOVE.B #40,BUFFER Definiere maximale Zeilenlänge

Kapitel 4

Vom Text zum Programm

Es sind mehrere Stufen zu überschreiten, ehe aus der Definition einer Aufgabe für den Computer ein einwandfrei ablaufendes Programm entsteht.

Die nachfolgende Skizze zeigt schematisch die Entwicklungsstufen eines Programms. Zu Beginn ist möglichst genau zu formulieren, welche Leistungen das beabsichtigte Programm auf dem Computer erbringen soll. Insbesondere wird man dabei berücksichtigen, in welcher Form Daten ein- und auszugeben sind, die das Programm benötigt bzw. die es erzeugt. Die nächste Entscheidung ist hinsichtlich der Programmiersprache zu fällen. In diesem Buche stellen wir den Gebrauch der Assemblersprache dar, mit der der kompakteste und am schnellsten ausführende Code für die Zentraleinheit, die CPU MC68000, erzeugt wird. Der Anwender könnte sich natürlich auch für eine andere "höhere" Sprache entscheiden. Soweit es sich dabei um Kompilersprachen handelt, deren Befehle in Maschinencode umgewandelt werden, liegen weitere Bearbeitungsstufen vor der hier dargestellten Assemblierung.

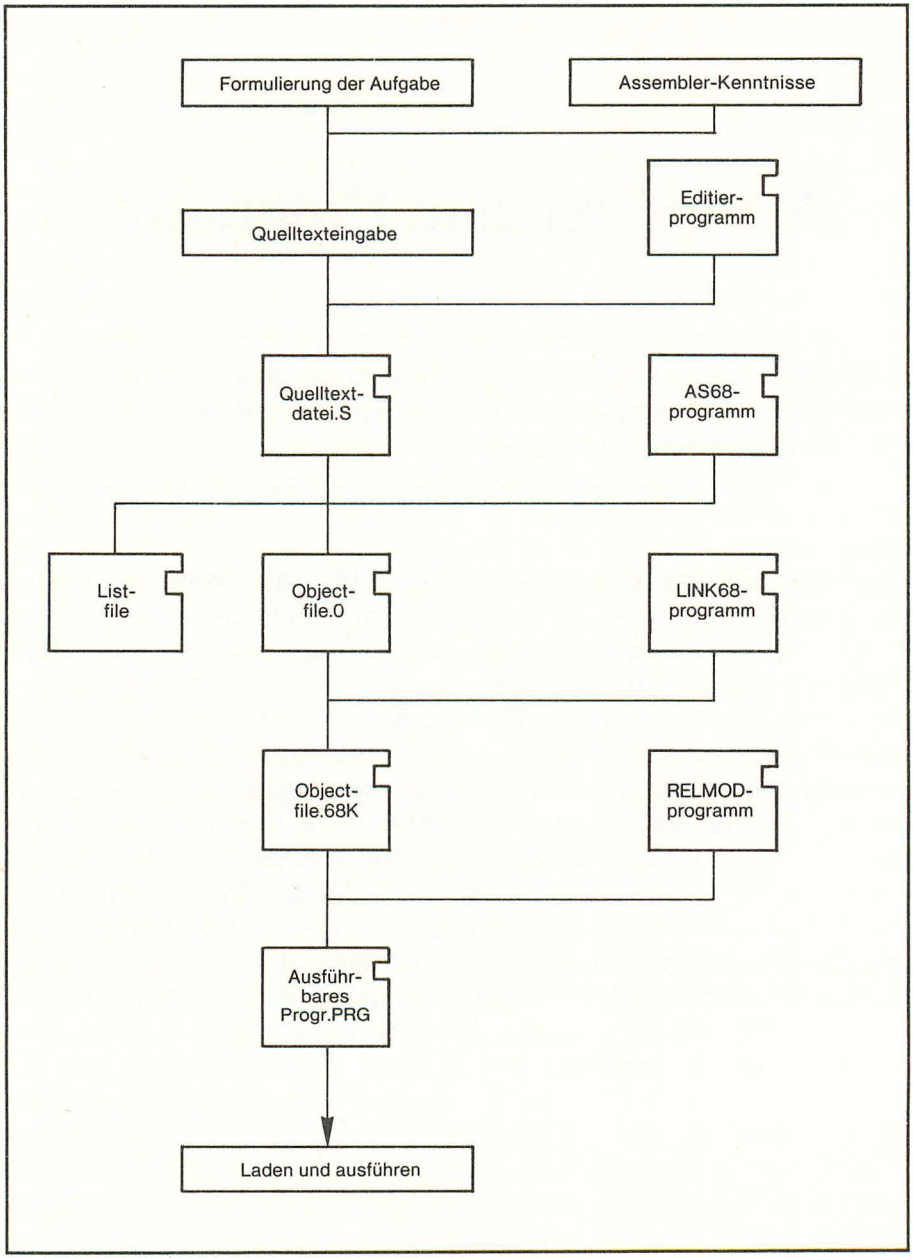


Abbildung 8: Vom Text zum Programm

Schreiben des Quelltextes

Programme werden als sogenannter Quelltext (Source) in Formulierungen der Assemblersprache in den Computer eingegeben. Für diese Sprache gelten bindende Regeln, die in diesem Buch dargestellt werden. – Für die Texteingabe muß in jedem Fall ein Textprogramm zur Verfügung stehen. Man nennt es auch Editor. In Abb. 8 ist es symbolisch als eine Diskette dargestellt. Das Textprogramm nimmt die Eingaben des Bedieners von der Tastatur entgegen, speichert sie vorübergehend im Computer, um sie spätestens am Ende der Sitzung auf Diskette auszugeben, und zwar als Quelltextdatei, die unter dem gewählten Namen später als Wiedereingabe in das Textprogramm (zur Korrektur und Ergänzung) oder als Eingabe für den Assembler benutzt wird. Auch diese Quelltextdatei ist in Abb. 8 als Diskette gezeichnet. – An anderer Stelle dieses Buches findet man Kriterien für die Auswahl eines leistungsfähigen Editierprogrammes.

Assemblierung

In der folgenden Stufe wird der Assembler mit Namen AS68 hereingerufen. Ihm wird mitgeteilt, welchen Namen die zu bearbeitende Quelltextdatei trägt. Zweck des Assemblers ist es, aus den Formulierungen des Quelltextes ein Ausgabefile zu erzeugen, das für die nachfolgenden Bearbeitungsstufen tauglich ist. Diese Ausgabedatei nennt man auch 'Objectfile'. Sie enthält neben dem Opcode der Maschinenbefehle Verschiebeinformation für das Linkerprogramm.

Eine wichtige Aufgabe des Assemblers ist daneben das Erzeugen der Assembler-Liste, in der im wesentlichen die Zuordnung des erzeugten Objectcodes zu den Formulierungen des Quelltextes zu finden ist. Auch diese Listung wird, sofern sie angefordert ist, als Text auf eine Diskettendatei abgelegt. Möglicher Bestandteil der Listung ist ferner eine Symboltafel, auf deren Bedeutung noch eingegangen wird. Das Assemblerprogramm weist schließlich auf Fehler im Quellprogramm hin.

Kopplung an Speicheradressen im Linker

Das vom Assembler erzeugte Objectfile ist noch keineswegs das lad- und ausführbare Maschinenprogramm. Es muß noch über die Stufen des Linkers und des Relocators geführt werden. Das Linkerprogramm mit Namen LINK68 hat die Aufgabe, ein oder mehrere unabhängig voneinander assemblierte Objectfiles miteinander zu verbinden und vor allem gegenseitige Bezüge in Adreßinformation aufzulösen. Die Ausgabedatei des Linkers wird schließlich vom Programm RELMOD auf absolute Adressen umgerechnet, und zwar in einer Form, die vom Betriebssystem des Computers, dem GEMDOS, akzeptiert wird.

Jede der genannten Stufen verlangt Anweisungen an den Computer, die für die jeweils benutzten Hilfsprogramme eigene Formulierungen und Parameter fordern Auch diese werden noch dargestellt. Für den praktischen Gebrauch schafft man sich ein Kommandofile, das die notwendigen Bearbeitungen standardmäßig abruft und damit für einen sehr automatischen Ablauf sorgt.

Speicher- und Zeitbedarf

Neben den eingezeichneten Dateien werden während einer Assemblierung vorübergehend auch Hilfsdateien angelegt. Der Leser erkennt, daß insgesamt viele Zugriffe auf Dateien erfolgen. Wenn diese nun auf Diskettenlaufwerken geführt werden, so entsteht einerseits ein Speicherbedarf, der schnell die Kapazität einer minimalen Laufwerksausrüstung überschreitet. In diesem Fall muß man mit Diskettentausch und Umkopierungen arbeiten. Das ist immer lästig und kann zu Fehlern führen. Andererseits ist mit vielen Diskettenzugriffen ein großer Zeitbedarf verbunden, den man in Hinblick auf höhere Produktivität vermeiden sollte. Der Ausweg liegt in der Installierung einer RAM-Disk, für die es auf den Atari-Rechnern bereits Programme zur Implementierung gibt. Je nach Ambition ist schließlich auch an den Anschluß eines schnellen Festplattenlaufwerkes zu denken.

Dateitypen

Beim Betriebssystem des Atari und auch bei seinem Assembler wurde eine Benutzeroberfläche gewählt, die historisch auf dem Betriebssystem CP/M fußt. Es sind daher einige Regeln aus dieser Umgebung zu beachten: Dateinamen auf den Disketten dürfen bis zu 8 Zeichen lang sein. Ihnen folgen optional ein Punkt und 1 – 3 Buchstaben, die den Dateityp kennzeichnen. Diesen Anhang nennt man auch File Extension. Die in diesem Abschnitt bezeichneten Hilfsprogramme fordern nun, daß die Dateien des Benutzers ganz spezifische Typbezeichner als Extension haben, die nachstehend verzeichnet werden. Bei anderen Assemblern als von DRI achte man auf deren Forderungen.

Editor-Ausgabe	Datei.S Dateityp 'S'	
Assembler-Eingabe	Datei.S	
Assembler-Ausgabe	Datei.0	
Linker-Eingabe	Datei.0	
Linker-Ausgabe	Datei.68K	
RELMOD-Eingabe	Datei.68K	
RELMOD-Ausgabe	Datei. PRG	Das ausführbare Programm

Kapitel 5

Leistungen eines Textprogrammes

Nicht nur beim Abfassen von Korrespondenztexten, sondern besonders auch für Eingabe und Edition von Quelltexten beim Programmieren hat das benutzte Textsystem großen Einfluß auf die erzielbare Leistung.

Wichtige Kriterien

Zunächst ist zu fordern, daß Tasteneingaben gleichlautend auf dem Bildschirm erscheinen. So selbstverständlich ist das gar nicht. Bei im Ausland entstandenen Textprogrammen ist die Annahme der deutschen Umlaute und ihre Darstellung auf dem Bildschirm wie auf dem Drucker keineswegs ein Standard. Und das gilt auch hinsichtlich der in Programmiersprachen häufig benutzten Sonderzeichen. Zu denken ist da an geschweifte und eckige Klammern oder an den Hochpfeil als Zeichen für Exponentiation. Daß der sog. Klammeraffe auf deutsch eingerichteten Computern gelegentlich als das Zeichen für Paragraph erscheint, ist noch gerne hinzunehmen.

Die abgespeicherten Zeichen müssen weiterhin eine hexadezimale Darstellung haben, die vom Assemblerprogramm gleichlautend ausgedeutet werden kann. Die klarsten Verhältnisse herrschen dann, wenn Zeichen grundsätzlich als ASCII-Zeichen

codiert sind. Es gibt Textsysteme, die mehr auf Korrespondenz und formatierten Druck ausgelegt sind, für die Steuerzeichen in den laufenden Text eingestreut werden. Eine solche Form der Abspeicherung ist als Eingabe für den Assembler ungeeignet. Man muß aber darauf hinweisen, daß einige Textsysteme eine Abspeicherung auch ohne solche Steuerinformation zulassen.

Ein wichtiges Kriterium sind die Bearbeitungsmöglichkeiten, die ein Editor bietet. Bei der laufenden Eingabe sollte man mindestens mit dem Cursor in eine offene Zeile hineinfahren können, um etwas einfügen und fortlassen zu können, ohne daß der Rest des Textes verändert wird. Besser ist noch die Möglichkeit, mit dem Cursor an jede Stelle des Bildschirmes fahren zu können, um Veränderungen vorzunehmen (Screen Editing).

Wichtig sind Funktionen, die ein Umsetzen von Zeilen und Textblöcken erlauben, so daß eine neue Reihenfolge im Text entsteht. Zeilen und Textblöcke sollten innerhalb des Textes kopierbar sein. Man sollte ferner Textblöcke aus schon vorhandenen anderen Dateien an beliebige Stelle einziehen können, um sich die Neueingabe identischer Texte ersparen zu können. Auch die umgekehrte Funktion sollte gegeben sein, daß man nämlich Textblöcke auf eine besondere Datei abspeichern kann. Von großer Bedeutung sind ferner Funktionen für das Suchen und Weitersuchen bestimmter Textketten sowie für das einmalige oder automatisch wiederholte Verändern von Strings. Auch sie sollten leicht zu bedienen sein.

Der Markt bietet inzwischen einige Textprogramme für den Atari 520ST, die den hier genannten Ansprüchen zum Teil mit zusätzlichem Komfort genügen.

Kapitel 6

Die typische Befehlszeile

Nach den Bildungsregeln der Assembler-Sprache hat eine Befehlszeile folgendes allgemeines Format ihrer möglichen Felder:

Label Befehl.Operationsbreite Ausdruck,Ausdruck Kommentar <CR>

Nicht alle vorstehenden Felder müssen in einer Befehlszeile vorhanden sein. Die Verwendung eines Labels und das Schreiben von Kommentaren sind nicht zwingend. Gleichwohl wird man von Bezeichnern (Labels) Gebrauch machen, wo immer es für die Bezeichnung von Einsprungstellen, Tabellen und Operanden sinnvoll ist, ebenso von reichlich Kommentaren für die Dokumentation. Ehe wir die Felder Befehl und Ausdruck ausführlicher angehen, sollen die einfacheren Felder einer Zeile abgehandelt sein: Das Zeilenende-Zeichen <CR> (Carriage Return) ist zwingender Abschluß einer Befehlszeile.

Label

Das Assembler-Programm muß Label als solche erkennen können. Daher gelten folgende Regeln: Wenn ein Label in der ersten Schreibstelle einer Zeile beginnt, dann ist sein natürlicher rechter Trenner gegen den Rest der Zeile ein Zwischenraum (Space). Beginnt eine Zeile jedoch mit einem oder mehreren

Spaces (das Label ist eingerückt), dann muß unmittelbar hinter dem letzten Zeichen des Namens ein Doppelpunkt folgen, z. B. also LABEL:
Dieser Doppelpunkt wird natürlich nicht Bestandteil des Namens, er ist nur Trenner.

Zur Schreibweise von Labeln: Sie bestehen aus einem oder mehreren gültigen Zeichen. Bis zu 8 Zeichen sind signifikant (werden unterscheidend erkannt). Gültige Zeichen sind die großen und kleinen Buchstaben A ... Z, die Ziffern 0 ... 9, die Zeichen $ (Dollar), . (Punkt) und _ (Unterstreichungsstrich). Der Assembler unterscheidet bei Labeln zwischen Groß- und Kleinschreibung.

Assembler-Definitionen, Symbolerklärungen

```
 1                                  * Assembler-Definitionen, Symbolerklaerungen
 2                                  *
 3                                  SYMBOL1 =    $1234              * hexadezimales Argument
 4                                  Symbol1 =    $4567              * hexadezimales Argument
 5                                  symbol2 equ %111                * binaeres Argument
 6                                  SYMBOL3 .equ @11                * oktales Argument
 7                                  Symbol4 = 10                    * dezimales Argument
 8                                  *
 9                                  * Ablegen von Konstanten:
10 00000000                         .data                           * Datensegment
11 00000000 01                      label1: .dc.b 1                 * Ein Byte ablegen
12 00000002                         .even                           * Ausrichtung an Wortgrenze
12 00000002
13 00000002 0001000200030004        label2: .dc.w 1,2,3,4,Symbol1 * mehrere Worte ablegen
13 0000000A 4567
14 0000000C 00001234                label3: .dc.l SYMBOL1           * Langwort
15 00000010 616263646566            string  .dc.b 'abcdef'          * Textkette ablegen
16 00000016 414243                          .dc.b "ABC"             * weitere Form fuer Textkette
17 00000019                         .end                            * Ende des Quelltextes

S y m b o l   T a b l e

SYMBOL1   00001234 ABS   SYMBOL3   00000009 ABS   Symbol1   00004567 ABS
Symbol4   0000000A ABS
label1    00000000 DATA  label2    00000002 DATA  label3    0000000C DATA
string    00000010 DATA  symbol2   00000007 ABS
```

Bewertung von Labeln

Wenn eine Zeile mit einem Label beginnt, so gibt es zwei Möglichkeiten für dessen Bewertung:

a) Ihm folgt eine der Direktiven .EQU oder .SET. Dann wird das Label mit dem Wert des Ausdruckes versehen, der rechts von der Direktive steht.
b) Das Label steht allein oder es folgt ihm ein Befehl oder Kommentar. Dann erhält es einen Wert, der dem gegenwärtigen Stand des Zuordnungszählers entspricht.

Spielarten der Bewertung, ob ein Label einen absoluten Wert hat oder einen relokativen, der sich relativ zum Beginn des Programmabschnittes rechnet, treten durch die Direktive ORG ein. Soweit sie wirksam ist, erhalten Label einen absoluten Wert.

Kommentare

Sie können an letzter Stelle in einer Befehlszeile hinter ggfs. notwendigen anderen Bestandteilen stehen, wenn sie von diesen durch mindestens einen Space getrennt sind. Die zweite Möglichkeit, Kommentare in ein Programm aufzunehmen, ist die Verwendung reiner Kommentarzeilen. Sie beginnen mit einem Stern. Wenn das zweite Zeichen jetzt nicht ein Gleichheitszeichen ist, dann wird die ganze Zeile als Kommentarzeile behandelt, die für den Prozeß der Umwandlung keinerlei Bedeutung hat.

Befehl

Wir unterscheiden zwei Arten von Befehlen: a) **Mnemonics**, d. h. mnemonische Abkürzungen für Befehle der CPU MC68000. Sie werden zu Binärcode umgewandelt und kommen zur Laufzeit

des Programms zur Abarbeitung. b) **Direktiven**, d. h. Pseudo-Befehle an das Assemblerprogramm. Sie dienen verschiedenen Steuerungen und Dispositionen, die zur Zeit der Assemblierung berücksichtigt werden. Direktiven nennt man auch Pseudo-Opcodes. Einige Direktiven führen auch zur Ablage von Code in das assemblierte Programm. – Den Befehlen sind eigene Abschnitte gewidmet. Befehle und Direktiven können in kleinen oder großen Buchstaben geschrieben sein, die der Assembler gleichrangig verarbeitet.

Operationsbreite

Die CPU MC68000 hat einen 16 Bit breiten Datenbus. Gleichwohl kann sie bei vielen Befehlen nicht nur auf Worte (16 Bit) zugreifen, sondern auch auf Bytes (8 Bit) und auf Langworte (32 Bit). Für solche Befehle gibt es Spielarten der Zugriffsbreite (siehe Tafeln zum Befehlssatz). Sie werden geschrieben, indem man unmittelbar an den mnemonischen Befehl einen Punkt anhängt und einen der folgenden Operatoren:

.L Langwortbefehl (32 Bit)
.W Wortbefehl (16 Bit)
.B Bytebefehl, auch kurze Verzweigung (8 Bit)

In anderen Assemblern sehen wir für kurze Verzweigungen beispielsweise auch die Schreibweise BRA.S. S steht dabei für "short" = kurz.

Ausdrücke

Zahlreiche Befehle benötigen keinerlei Angabe eines Operanden oder Ausdrucks, an dem der Befehl zu vollziehen ist. Man denke an RTS (Return from Subroutine) oder an TRAPV (Ausnahmebedingung bei Überlauf des Operanden). Andere Befehle brauchen

nur einen Operanden, z. B. TRAP #14. Die meisten Befehle fordern jedoch die Angabe von zwei Operanden, weil der MC68000 eine Zweiadreß-Maschine ist. Beispiele sind MOVE.L D0,$5000 (Kopiere ein Langwort aus dem Adreßregister D0 an die vier Speicherstellen ab Adresse $5000). Wo zwei Ausdrücke erforderlich sind, werden sie durch ein Komma getrennt. Zwischenraumzeichen zwischen ihnen und dem Komma sind unzulässig.

Für die Bildung von Ausdrücken werden wir in den Beispielen dieses Buches noch viele Möglichkeiten sehen. Grundsätzlich läßt sich sagen, daß ein Ausdruck sich in irgendeiner Form aus Zahlen, Zeichen, Symbolen oder Registerlisten bewerten lassen muß, wobei auch eine arithmetische oder logische Verknüpfung zwischen Operanden in Betracht kommt.

Zahlen

Zahlen können Bestandteil von Ausdrücken sein. Der Assembler erkennt Zahlen zur Zahlenbasis 2, 8, 10 und 16, für die besondere Schreibweisen einzuhalten sind:

Dezimal	Ein String dezimaler Ziffern 0 ... 9 ohne Zusätze
Hexadezimal	Ein mit dem Dollar beginnender String aus dem Wertebereich folgender Symbole: 0 ... 9 und A ... F, z. B. $FF3456
Oktal	Ein mit dem Klammeraffen beginnender String aus Ziffern mit 0 ... 7, z. B. @732
Binär	Ein mit dem Prozentzeichen beginnender String mit Ziffern von 0 oder 1, z. B. %1011 = dezimal 11.

Register und Registerlisten

Auch Register und Registerlisten können in einer mnemonischen Form erklärt werden. Der Assembler erkennt folgende Schreibweisen in Groß- oder Kleinbuchstaben zur Bezeichnung von Registern:

r0 . . . r15	wobei r0 ... r7 den Datenregistern D0 ... D7
R0 . . . R15	entsprechen und r8 ... r15 den Adreßregistern A0 ... A7.
d0 . . . d7	und D0 ... D7 entsprechen den Datenregistern D0 ... D7 und
a0 . . . a7	und A0 ... A7 entsprechen den Adreßregistern A0 ... A7.
A7, Sp	werden als Ansprache des jeweils gültigen Stackpointers erkannt.

Registerbezeichnungen in der vorgenannten mnemonischen Schreibweise für einzelne Register treten in sehr vielen Befehlen auf. Registerlisten dagegen werden in Zusammenhang mit dem Befehl MOVEM benutzt: Übertrage mehrere Register (Move Multiple). Der Opcode des Befehles enthält dabei eine binäre Strichliste, welche Register zu transportieren sind. Und diese Strich- oder Registerliste läßt sich dem Assembler mit der Direktive REG symbolisch erklären. Dafür einige Beispiele:

Registerlisten definieren

```
 1                                  * Registerlisten definieren
 2                                  *
 3                                  stapel1: reg a0-a6            * Register a0-a6
 4                                  stapel2: reg d0-d7            * Register d0-d7
 5                                  stapel3: reg a5,a6,d0-d3,d6   * Einzelne Register und Register von-bis
 6                                  *
 7 00000000                         .text                         * Benutzung der Definitionen im Programm
 8 00000000 48E700FE                movem.l stapel1,-(a7)         * Bringe Liste nach stapel1 auf den Stack
 9 00000004 48A7FF00                movem.w stapel2,-(a7)         * Dito nach Liste Stapel2, Wortbreite
10 00000008 4C9F00FF                movem.w (a7)+,stapel2         * Register vom Stack zurueckladen
11                                  *
12 0000000C                         .end

S y m b o l   T a b l e

stapel1   000000FE ABS   stapel2   FFFFFF00 ABS   stapel3   FFFFF206 ABS
```

Bei der Niederschrift des Strings für die Liste sind das Komma und der Schrägstrich als Trenner zwischen einzelnen Aufgaben und Blöcken erlaubt; das Minuszeichen bestimmt Bereiche, z. B. D0 – D7 bedeutet: Register D0 ... D7.

Operatoren in Ausdrücken

Zwischen einzelnen Elementen eines Ausdruckes sind arithmetische und logische Verknüpfungen möglich, z. B. LABEL1+LABEL2. Das Pluszeichen ist dabei der Operator. Der vorstehende zusammengesetzte Ausdruck hat damit den Wert der Summe beider Label. Weitere arithmetische Operatoren sind –, *, / für Subtraktion, Multiplikation und Division mit ganzzahligem Ergebnis. Ein Minuszeichen kann auch unär als Vorzeichen eines Ausdruckes benutzt werden, z. B. –LABEL.

Zwischen Operanden ist auch eine logische Verknüpfung möglich, und zwar mit '&' für UND und '!' für ODER. So ist z. B. der Wert von $1!$8 die Zahl $9 oder $9&$1 ergibt $1.

Operanden können jedoch auch in einer gegeneinander verschobenen Weise miteinander verknüpft werden. Dem dienen die Operatoren '>>' und '<<' für die Rechtsverschiebung des linken Operanden (er wird dabei von links mit binären Nullen aufgefüllt) und für die entsprechende Linksverschiebung.

ASCII-Strings

Auch Ketten von einem oder mehreren ASCII-Zeichen können in Ausdrücken auftreten. Die Anwendung wird bei den Direktiven und in den Beispielen erklärt.

DAS SPRACHRESERVOIR

DAS SPRACHRESERVOIR

Kapitel 7

Die Befehlsgruppen

In diesem Buch werden die Befehle des 68000 in 14 Funktionsgruppen gegliedert und einzeln erklärt. Hier zeigen wir zunächst die Rolle der Hauptregister d. h. der je 8 Daten- und Adreßregister, und des Status. Befehle lassen sich aber auch anders gliedern:

Befehle ohne Operanden, Befehle mit 1 Operanden, Befehle mit 2 Operanden

Viele Betreiber anderer Computer haben dem Autor berichtet, daß sie noch keinen richtigen Durchblick bei der Programmierung des 68000 hätten. Es dürfte die Fülle der Möglichkeiten sein, die zunächst etwas ratlos macht. Wir haben daher die Befehle in 14 Funktionsgruppen gegliedert und besprechen dort jeweils die Besonderheiten. Diese Abschnitte mögen für die tägliche Arbeit als Referenz dienen. Wir sehen aber auch: Der Befehlssatz ist in der gegliederten Form gut übersehbar. Was müssen wir nun bei der Programmierung berücksichtigen?

Zunächst zur Hardware:

Die 8 Datenregister der CPU sind von allen Registern mit den größten Fähigkeiten zum Rechnen und zum logischen und bit-

weisen Verändern von Operanden versehen. Man benutzt sie daher vorwiegend zum Bearbeiten von Operanden.

Die 8 Adreßregister dienen vor allem als indirekte Zeiger auf Speicheradressen. Sie sind daher nur für einfache Adreßrechnungen eingerichtet, die den Status des Prozessors (Übertrag, Negativ-Bit usw.) nicht störend verändern.

Das Adreßregister A7 ist jeweils Stackpointer (Supervisor oder Anwender), der mit Beschickungen des Stacks oder Entnahmen von dort immer automatisch richtig nachgezogen wird. – Die im Stackpointer enthaltene Adresse zeigt jeweils auf das letzte Item, das auf dem Stack abgelegt wurde. Mit jeder Beschickung des Stacks verändert sich der Stackpointer von hohen zu niedrigeren Adressen. Umgekehrt bei Entnahmen.

Beim Laden von Adreßregistern ist die mögliche Vorzeichenerweiterung von Wortoperanden zu beachten. Die Spezialbefehle für Adreßregister wurden daher gesondert ausgewiesen.

Alle insgesamt 16 Daten- und Adreßregister können bei einigen Adressierungsarten auch als Indexregister dienen.

Operationen in und zwischen den Registern der Zentraleinheit laufen schneller ab, als vergleichbare mit Operanden im Speicher, weil das Laden von Operanden und das Abspeichern von Ergebnissen entfällt. Das Design der CPU ist daher auf ein Bearbeiten in den Datenregistern angelegt. Mit den Adreßregistern werden Speicheradressen verwaltet.

Das Bedingungscode-Register

Bei allen Entscheidungen im rein rechnenden und logischen Bereich spielt nur das Bedingungscode-Register (CCR) im Status (SR) mit. Hier sind 5 Bedingungsflags im Spiel, die abhängig von den Befehlen und ihren Ergebnissen gesetzt werden. Das CCR

ist dabei das niederwertige Byte des Statuswortes. – Die Tabellen zu den Befehlsgruppen weisen daher besonders die Beeinflussung der Flags oder die Abhängigkeit von Bearbeitungen von den Flags aus.

Das höherwertige Byte des Statuswortes kann nur im privilegierten Zugriff des Betriebssystems erreicht werden. Seine Bits regeln den Modus (Supervisor oder Anwender), den Einzelschritt-Modus (Trace) und die Bestimmung der Interrupt-Priorität. – Wenn die CPU im Anwendermodus ist und wenn dabei Befehle benutzt werden, die für den Supervisor-Modus privilegiert sind, dann tritt eine Ausnahmebedingung ein. Das Anwenderprogramm wird unterbrochen, die CPU lädt den Ausnahmevektor Nr. 8, und es wird die vom Betriebssystem vorgesehene Ausnahmeverarbeitung durchgeführt.

Befehle ohne Operanden

Viele Befehle bearbeiten keinerlei Operanden, so z. B. die Verzweigungen oder Sprünge. Sie verändern auch nicht den Status. Ihnen brauchen wir nur ein Ziel mitzuteilen. Andere haben automatisch ein Ziel. Man denke z. B. an RTS und RTE, Rückkehr von einem Unterprogramm oder von einer Ausnahmeverarbeitung.

Befehle mit 1 Operanden

Einige Befehle bearbeiten jeweils nur 1 Operanden. Man denke an TST (Prüfung auf Vorzeichen und Null) oder an die Verschiebe- und Rotierbefehle und einige andere.

Befehle mit 2 Operanden

Der 68000 ist eine Zweiadreß-Maschine. Die meisten Befehle fordern 2 Operanden. In der Assembler-Schreibweise steht der Quelloperand vor dem Komma, dahinter (ohne Space!) der Zieloperand, der auch das Ergebnis aufnimmt. Englisch nennt sich das Source und Destination. – In dieser Gruppe finden wir vor allem die arithmetischen und logischen Befehle sowie die Transporte "von-nach".

Die Arbeitsbreite der Befehle

Der Leser wird bemerkt haben, daß viele Befehle der CPU "eine Verarbeitung von Bytes, Worten und Langworten" zulassen. Auch die hier gegebene Vielfalt mag zunächst etwas verwirren. Daher: Bei vielen Befehlen ist die Arbeitsbreite automatisch impliziert, z. B. bei Multiplikation und Division oder bei den Befehlen zum Statusregister. Hier erzeugt der Assembler automatisch den richtigen Code. In den Tabellen wird das in der Spalte "Operanden" dadurch kenntlich gemacht, daß dort ein Wert in Klammern steht, z. B (32 Bit) oder (16 Bit).

Bei anderen Befehlen schreiben wir direkt hinter die mnemonische Befehlsabkürzung einen Punkt und dann einen der Buchstaben L, W oder B. Es bedeuten dabei **.L** = Langwort (32 Bit), **.W** = Wort (16 Bit) und **.B** = Byte (8 Bit). Gelegentlich haben wir .W* geschrieben. Der Stern darf dabei nicht in Quelltexte des Assemblers übernommen werden. Er dient Ihnen als Hinweis, daß bei Verarbeitung in Wortbreite eine Vorzeichenausdehnung stattfinden kann.

Welche Arbeitsbreite soll man nun wählen? Über den Daumen kann man folgenden Hinweis geben: Ein- und Ausgaben von Text erfolgen zeichenweise. Bei entsprechenden Routinen werden wir also die Breite Byte (.B) wählen. Auch die dezimalen Rechenarten arbeiten byte-weise.

Die Register von Interfacebausteinen werden zumeist byteweise angesprochen. Beim Arbeiten in Wortbreite können Störwirkungen auftreten.

Viele Programme enthalten Flag-Variablen, in denen logische Bedingungen gemerkt werden. Auch bei ihnen wird meistens eine Verankerung und Ansprache in Byte-Größe ausreichen.

Bei binären Berechnungen hängt die Wahl von den größten vorkommenden Werten ab. Bei sehr großen Werten oder hoher Genauigkeit wird man sogar Datenfelder vorsehen, die aus vielleicht zwei Langworten bestehen.

Adressen, die für die Ansprache von Operanden im Speicher zu bestimmen sind, wird man grundsätzlich in Langworten berechnen.

Direktiven an das Assembler-Programm erlauben es, die Arbeitsbreite der Befehle über längere Passagen festzulegen, so daß man nicht immer die vorgenannten Zusätze .L, .W oder .B schreiben muß.

Das Extendbit

Dem Newcomer macht die Bedeutung das Extendbit (X) im Status manchmal gedankliche Schwierigkeiten. Zunächst: Soweit durch Befehle geändert, wird es identisch wie das Carry-Bit gesetzt. Zweitens: Für das Addieren, Subtrahieren und Verschieben von Datenfeldern gibt es jeweils zwei Befehle. Derjenige ohne 'X' im Namen wird auf den niederwertigsten Teil des Datenfeldes angewandt. Die höheren Teile des Datenfeldes werden mit dem Befehl behandelt, der das 'X' im Namen hat. Dabei werden Überträge oder das Borgen aus vorherigen Operationen berücksichtigt.

Befehle mit Extendbit-Verarbeitung sind ADDX, NEGX, SUBX, ROXL und ROXR.

Andere Befehlsgruppierungen

Viele Befehle mit Direktoperand (Konstante) haben ein 'I' am Namensende. Wir erwähnen:
ADDI, SUBI, ANDI, EORI, ORI, CMPI.

Schnell ausführende Befehle mit kleinen Konstanten im Wertebereich 1 ... 8 sind:
MOVEQ, ADDQ, SUBQ.

BCD-Befehle haben "BCD" im Namen: ABCD, NBCD, SBCD.

Zur Beachtung: Beim 68000 werden in den Befehlen enthaltene Adressen in der Reihenfolge von High- zu Low-Bytes abgespeichert (umgekehrt also wie z. B. beim 6502/65816).

Kapitel 8

Datenübertragungen

Der allgemeine Transportbefehl ist MOVE. Er kann für die Datenquelle alle, für das Datenziel fast alle Adressierungsarten des 68000 benutzen. Ziele, die relativ zum Programmzähler adressiert sind, unterliegen Einschränkungen.

MOVEQ führt für kleine Konstanten schneller aus als MOVE.

MOVEP dient dem Datenverkehr mit den synchronen Peripheriebausteinen der Familien 68xx und 65xx, die eine besondere elektrische Anbindung erfordern.

MOVEM wird zur Sicherung von bis zu 16 Registerinhalten mittels eines Befehles benutzt, die andere Routinen (das Betriebssystem) später unversehrt zurückerhalten müssen. Je nach gewählter Adressierungsart treten Besonderheiten ein, die man beachten muß:

MOVEM wird typisch paarweise, d. h. am Anfang und Ende eines Erledigungsblockes, eingesetzt: Man sichert zunächst eine Registerliste in den Speicher und holt sie später identisch zurück. Wenn man die folgenden typischen Adressierungsarten beachtet, braucht man sich weiter um nichts zu kümmern:

MOVEM Registerliste, –(An) macht erst Platz im Stack für das erste Wort und speichert die Registerinhalte zu fallenden Adressen hin, und zwar in der getroffenen Auswahl ab A7 bis hin zu A0 und dann D7 bis hin zu D0. (An) zeigt am Schluß auf das zuletzt abgelegte Wort.

Beim Zurückladen kehrt sich alles um:

MOVEM Registerliste, (An)+ (Postinkrement) holt Worte aus dem Speicher ab Adresse in (An) und legt sie in die Register, ggfs. Auswahl von D0 ... A7, ab. (An) wird wegen (An)+ laufend um 2 oder 4 erhöht, je nach .W oder .L. Zum Schluß hat (An) als Stapelzeiger wieder den Wert wie vor der ganzen Aktion.

Bei den übrigen Adressierungsarten für MOVEM wird ab der im Befehl angegebenen Basisadresse automatisch hin zu höheren Adressen gearbeitet, und zwar gleichermaßen beim Speichern und Holen. Die Registerliste wird dabei in der Reihenfolge (auch in Auswahl) D0 ... D7 bedient. (Man beachte beim Zurückladen in die Register und wenn die Operationsbreite .W gewählt ist, daß die geholten Worte in alle Register vorzeichenerweitert geladen werden).

Tabelle 1: Datenübertragung

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
EXG	(32 Bit)	Rx, Ry							Inhalt zweier Register vertauschen
MOVE	.L .W .B	<ea>, <ea>			N	Z	0	0	Übertrage einen Operanden von der Quelle zum Ziel
MOVEM	.L .W*	Registerliste, <ea> <ea>, Registerliste							Übertrage mehrere Registerinhalte (multipel) lt. Registerliste zum/vom Speicher/Stackbereich
MOVEP	.L .W	Dx, d(Ay) d(Ay), Dx							Operandenübertragung speziell zu/von 8-Bit-Peripheriebausteinen der Typen 68xx
MOVEQ	L*	#Konstante, Dn			N	Z	0	0	Konstante in ein Datenregister laden Wertebereich –128 bis +127 Vorzeichenerweiterung auf 32 Bit
SWAP	(16/16 Bit)	Dn			N	Z	0	0	Die beiden Hälften eines Datenregisters vertauschen

ANMERKUNG: L* bzw. W* bedeuten Vorzeichenerweiterung auf 32 Bit

Kapitel 9

Spezialbefehle für Adreßregister

Die Inhalte von Adreßregistern werden vor allem als Zeiger auf Operanden im Speicher benutzt, wie z. B das Register HL im Z80 oder die Pointer in der Zero Page des 6502. Dabei kann man diese Zeiger als effektive Adresse betrachten oder sie bei der Adressenbildung auch mit einem Versatz (Displacement) und/oder einem Index versehen, damit sie auf bestimmte Elemente z. B. in einem Datenfeld zeigen.

Bei der Benutzung von Wortoperanden für Adreßregister beachte man, daß diese auf 32 Bit vorzeichenerweitert behandelt werden.

LEA berechnet eine Adresse nach den zugelassenen Bildungsregeln, sendet sie jedoch nicht auf den Adreßbus, sondern lädt sie z. B. als eine Basis in ein Adreßregister.

PEA, der korrespondierende Befehl, bringt eine wie bei LEA bestimmte Adresse auf den Stack.

Adreßregister A7 (SP) enthält je nach S-Bit im Status entweder den System-oder den Anwender-Stackpointer (zwei getrennte Register in der Hardware, die mit dem S-Bit umgeschaltet werden). Aber auch jedes andere Adreßregister kann als ein Stackpointer benutzt werden, um z. B. einen Datenstack aufzu-

bauen. Man wählt dann bei Beschickungen die Adressierungsart ,–(An) für das Ziel und umgekehrt bei Entnahmen (An)+, für die Quelle.

Adreßregister haben ferner eine Bedeutung bei den Befehlen LINK und UNLK zur Bereichsreservierung (Frame) auf dem Stack für Unterprogramme und für MOVE USP.

Tabelle 2: Spezialbefehle für Adreßregister

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
LEA	(32 Bit)	<ea>, An							Eine effektive Adresse in ein Adreßregister laden (immer 32 Bit)
MOVEA	.L .W*	<ea>, An							Adreßregister mit dem Inhalt des Quelloperanden laden
ADDA	.L .W*	<ea>, An							Binäre Addition eines Operanden in ein Adreßregister (Ein Übertrag-X-Bit hat dabei keinen Einfluß)
CMPA	.L .W*	<ea>, An			N	Z	V	C	Vergleich eines Operanden mit einem Adreßregister. Das Bedingungsregister wird gesetzt.
SUBA	.L .W*	<ea>, An							Subtraktion eines Operanden von einem Adreßregister (Das X-Bit-Borgen hat keinen Einfluß)

ANMERKUNG: Bei Wortbreite des Quelloperanden wird sein Vorzeichen-Bit 15 in die Bits 16–31 ausgedehnt. Hier kann eine Falle lauern!

W* = automatische Vorzeichenerweiterung

Kapitel 10

Statusbefehle

Der Prozessorstatus (SR) enthält im höherwertigen Byte Systeminformationen, die nur im Supervisor-Modus verändert werden können. 'MOVE to SR' ist daher ein privilegierter Befehl.

Das niederwertige Byte im Status ist das Bedingungscode-Register (CCR), bei dem nur die niedersten 5 Bits per Hardware benutzt sind. Die Mehrzahl der Befehle führt, abhängig von den Ergebnissen ihrer Ausführung, zu Veränderungen in diesen 5 Bits, die damit eine zentrale Bedeutung für die Entscheidungslogik haben.

Es gibt keine mnemonischen Befehle für das Setzen und Löschen einzelner Bits (Flags) im Status, die z. B. CLC oder SEC hießen. Es wird vielmehr mit dem ORI-Befehl zum Setzen lt. Muster und mit ANDI zum Löschen von Bits lt. Muster gearbeitet.

Im Gegensatz zu 'MOVE to SR' ist der Befehl 'MOVE from SR' nicht privilegiert. Man kann den Status also (auch per Anwenderprogramm) als ein Wort in den Speicher oder, zur Sicherung, auf den Stack bringen und nun ggfs. einzelne Bedingungen prüfen. Normalerweise macht man jedoch von den bedingten Verzweigungsbefehlen Gebrauch.

Eine wichtige Einrichtung sind die besonders besprochenen Befehle der Gruppe Scc: 'Set on Condition', setze ein Flag-Byte

abhängig von einer Bedingung im Status. Mit ihnen kann sich das Programm Bedingungen zur späteren Verwertung merken.

Tabelle 3: Statusbefehle

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
ANDI to CCR	(16 = 5 Bit)	#Konstante, CCR		X	N	Z	V	C	Maskieren des Bedingungscode-Registers mit den niederwertigsten 5 Bits einer Wortkonstanten
ORI to CCR	(16 = 5 Bit)	#Konstante, CCR		X	N	Z	V	C	Hinzu-ODERn von Bits im Bedingungscode-Register gemäß den 5 niederwertigsten Bits einer Wortkonstanten
MOVE to CCR	(16 = 5 Bit)	<ea>, CCR		X	N	Z	V	C	Einen Wortoperanden in das Bedingungscode-Register übertragen. Nur die niederwertigen 5 Bits wirken sich aus.
ANDI to SR	(16 Bit)	#Konstante, SR	S	X	N	Z	V	C	Privilegierter Befehl! Im gesamten Statuswort Bits lt. Konstante zu 0 setzen.
ORI to SR	(16 Bit)	#Konstante, SR	S	X	N	Z	V	C	Privilegierter Befehl! Im gesamten Statuswort Bits lt. Konstante Hinzu-ODERn.
MOVE to SR	(16 Bit)	<ea>, SR	S	X	N	Z	V	C	Privilegierter Befehl! Ein ganzes Wort in das Statusregister übertragen.
MOVE from SR	(16 Bit)	SR, <ea>							Übertragung des gesamten Statuswortes an effektive Adresse

ANMERKUNG: Wenn ein privilegierter Befehl im Anwender-Modus benutzt wird, so führt das zu einer Exception mit Vektor Nr. 8 wegen Privilegverletzung.

Kapitel 11

Status prüfen und merken

Die Befehle der Gruppe Scc (setze ein Byte in Abhängigkeit von einer Bedingung im Bedingungscode-Register CCR) arbeiten automatisch byte-weise an der effektiven Adresse. Sie dienen dem Merken eingetretener Bedingungen bis zu einer späteren Entscheidung. Trifft die Bedingung zu, so wird das Byte zu $FF gesetzt (ist also negativ), andernfalls zu $00.

Die meisten Befehle verändern das Bedingungscode-Register im Status. Verzweigungsbefehle, auch solche der Gruppe DBcc für Schleifenkontrukte, nehmen jedoch auf den aktuellen Status Bezug. Entscheidungen müssen jedoch oft in Abhängigkeit von früher eingetretenen Bedingungen gefällt werden. Die Befehlsgruppe Scc setzt zu diesem Zweck Flag-Bytes (in Datenregistern oder im Speicher), die dann später z. B. mit dem Befehl TST wieder abgefragt werden können.

Der Status könnte natürlich auch als ein Wort in den Speicher gebracht werden, wo man ihn dann später mit Bit-Befehlen prüft. Dieses Verfahren wäre umständlicher als dasjenige mit den Befehlen Scc, zumal auch das Gespeicherte nur privilegiert in den Status zurückgeladen werden könnte.

Tabelle 4: Status prüfen und merken

Befehl	Operanden	Syntax, Quelle, Ziel	Statusabhängigkeit						Befehlswirkung: Setze Byte zu $FF, wenn ... bzw. zu $00, wenn nicht ...
				X	N	Z	V	C	
SCC SCS	(8 Bit)	<ea>						C	Carry Clear Carry Set
SEQ SNE	(8 Bit)	<ea>				Z			Gleich Ungleich
SHI SLS	(8 Bit)	<ea>				Z		C	Größer Kleiner oder gleich
SPL SMI	(8 Bit)	<ea>			N				Positiv Negativ
SVC *) SVS *)	(8 Bit)	<ea>					V		V-Flag gelöscht V-Flag gesetzt
SGE *) SLT *)	(8 Bit)	<ea>			N		V		Größer oder gleich Kleiner als
SGT *) SLE *)	(8 Bit)	<ea>			N	Z	V		Größer Kleiner oder gleich
ST SF	(8 Bit)	<ea>							Immer (True) Niemals (False)

ANMERKUNG: Mit *) bezeichnete Befehle arbeiten mit der Arithmetik für Zweierkomplement zusammen.

Kapitel 12

Binäre Rechenarten

Der 68000 kann die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division binär ausführen. Die Datenregister haben dabei eine bevorzugte Stellung. Bei der Addition und Subtraktion größerer Zahlen muß man auf Überträge oder "Borgen" im Extendbit (X) achten. Nach einer Addition zeigt X = 1 einen Übertrag an, der mit ADDX auf weitere Stellen des Datenfeldes richtig übertragen wird. Nach einer Subtraktion zeigt X = 1 ein Borgen aus der Vorstelle an, was beim Befehl SUBX berücksichtigt wird.

CLR löscht einen Operanden (speziell den Zieloperanden) zu Null.

Kleine Direktwerte addiert und subtrahiert man mit den schneller ausführenden Befehlen **ADDQ** (quick) und **SUBQ**.

Wenn man die Befehle **ADDX** und **SUBX** ("mit Extendbit") auf Datenfelder im Speicher anwendet, dann muß man

a) auf das Predrekrement bei Quelle und Ziel achten: –(Ax), –(Ay). Die Adressierung ist also "rechts" vom 1. Operanden zu beginnen.

b) Vor der Bearbeitung des 1. Operanden im Feld ist das X-Bit im Status mit dem ANDI to CCR auf 0 zu setzen. Man richtet ferner einen Hilfszähler möglichst in einem

Tabelle 5: Binäre Rechenarten (1)

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
ADD	.L .W .B	<ea>, Dn Dn, <ea>		C	N	Z	V	C	Addition ohne Übertrag (X-Bit ohne Einfluß) unter Beteiligung eines Datenregisters
ADDI	.L .W .B	#Konstante, <ea>		C	N	Z	V	C	Addition einer Konstanten ohne Übertrag (aus dem X-Bit) zum Zieloperanden
ADDQ	.L .W .B	#Konstante, <ea>		C	N	Z	V	C	Schnelle Addition einer kleinen Konstanten (Wertebereich 1–8) zum Zieloperanden
ADDX	.L .W .B	Dy, Dx –(Ax), –(Ay)		C	N	Z?	V	C	Addition mit Übertrag (aus dem X-Bit) zum Zieloperanden. Das Z-Flag bleibt unverändert, wenn das Ergebnis 00 ist.
CLR	.L .W .B	<ea>			0	1	0	0	Lösche den Inhalt eines Operanden zu Null.
DIVS	(32/16 Bit)	<ea>, Dn			N?	Z?	V	0	Division eines vorzeichenbehafteten Dividenden in einem Datenregister durch einen 16-Bit vorzeichenbehafteten Divisor, Quotient im Datenregister
DIVU	(32/16 Bit)	<ea>, Dn			N?	Z?	V	0	Wie DIVS. Die Operanden werden jedoch als vorzeichenbehaftete Zahlen behandelt.
EXT	.L* .W*	Dn			N	Z	0	0	Ausdehnung des Vorzeichens (in Bit 7 bzw. in Bit 15) auf Wort- oder Langwortlänge in einem Datenregister.

ANMERKUNG: L* bzw. W* bedeuten Vorzeichenerweiterung auf 32 Bit.

Tabelle 6: Binäre Rechenarten (2)

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
MULS	(16/32 Bit)	<ea>, Dn			N	Z	0	0	Vorzeichenbehaftete Multiplikation zweier 16-Bit-Operanden (davon mindestens einer in einem Datenregister Dn). Ergebnis 32 Bit im Datenregister Dn.
MULU	(16/32 Bit)	<ea>, Dn			N	Z	0	0	Multiplikation wie MULS. Die Operanden werden als vorzeichenfrei behandelt.
NEG	.L .W .B	<ea>		C	N	Z	V	C	Zweierkomplement einer Zahl an effektiver Adresse bilden. (Subtraktion: 0 – Operand → Operand).
NEGX	.L .W .B	<ea>		C	N	Z	V	C	Zweierkomplement einer Zahl unter Berücksichtigung vorherigen Borgens (X-Bit) an effektiver Adresse bilden. (Für Feldoperationen mit großen Zahlen.)
SUB	.L .W .B	<ea>, Dn Dn, <ea>		C	N	Z	V	C	Subtraktion des Quelloperanden vom Zieloperanden unter Beteiligung eines Datenregisters, ohne Berücksichtigung vorherigen Borgens (X-Bit).
SUBI	.L .W .B	#Konstante, <ea>		C	N	Z	V	C	Eine Konstante vom Zieloperanden ohne Berücksichtigung vorherigen Borgens (X-Bit) subtrahieren.
SUBQ	.L .W .B	#Konstante, <ea>		C	N	Z	V	C	Einen kleinen Direktoperanden im Wertebereich 1–8 vom Zieloperanden ohne Berücksichtigung vorherigen Borgens (X-Bit) subtrahieren.
SUBX	.L .W .B	Dy, Dx –(Ax), –(Ay)		C	N	Z?	V	C	Den Quelloperanden unter Berücksichtigung vorherigen Borgens (X-Bit) vom Zieloperanden subtrahieren. (Für Operationen mit großen Zahlen.) Das Z-Flag bleibt unverändert, wenn das Ergebnis Null ist.

Register ein, der mit der Zahl der Schritte vorgeladen wird. Der Rest läuft in einer Bearbeitungsschleife automatisch.

Multiplikationen und Divisionen erfolgen immer in 16 x 16 = 32 Bit bzw. in 32/16 = 16 Bit, und zwar unter Beteiligung eines Datenregisters. Stehen beide Operanden in einem Datenregister, so hat man die schnellere Ausführung.

Kapitel 13

Dezimale Rechenarten

Beim 68000 sind die dezimalen Rechenarten mit Addition und Subtraktion nur wenig unterstützt. Sie arbeiten grundsätzlich byteweise. Das im Ergebnis Null anzeigende Z-Flag des CCR arbeitet hier nur erwartungsgemäß, wenn man es vor der Operation auf '1' gesetzt hat (ORI to CCR). Ist es nämlich vorher '0', so bleibt es auch dann '0', wenn es dem Ergebnis nach eigentlich '1' sein müßte.

Tabelle 7: Dezimale Rechenarten

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
ABCD	.B	Dy, Dy –(Ay), –(Ax)		C	?	Z?	?	C	Dezimale Addition, byteweise, unter Einbeziehung des Übertrages (X-Bit)
NBCD	.B	<ea>		C	?	Z?	?	C	Negation von BCD-Zahlen, byteweise, Bildung des 9er- oder 10er-Komplements je nach X-Bit = 1 oder = 0
SBCD	.B	Dx, Dy –(Ax), –(Ay)		C	?	Z?	?	C	Dezimale Subtraktion, byteweise, Zieloperand – Quelloperand unter Einbeziehung des Borgens (X-Bit)

ANMERKUNG: Das Z-Flag im Status spiegelt nur dann den Wert des Ergebnisses = 0 wider, wenn es vor der Operation zu "1" gesetzt war; es spiegelt aber immer ein Ergebnis $\neq$ 0 wider.

Kapitel 14

Logische Befehle

Wir haben die logischen Grundverknüpfungen mit UND, EXKLUSIV ODER, ODER auch mit Direktwerten. Die Struktur der Befehle folgt denen für das binäre Rechnen. Wir brauchen uns jedoch nicht um das Carry- oder Extendbit zu kümmern.

Bei der Benutzung der logischen Befehle prüfe man auch immer, ob man nicht mit Befehlen für das Prüfen, Setzen und Löschen einzelner Bits (Bitmanipulation) etwas übersichtlichere Verhältnisse schaffen kann. Diese nimmt man bevorzugt für einfache Prüfungen und Umschaltungen. Die logischen Befehle sind für die Prüfung und Abgabe komplexerer Muster zu bevorzugen.

Tabelle 8: Logische Befehle

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
AND	.L .W .B	<ea>, Dn Dn, <ea>			N	Z	0	0	Logisches UND unter Beteiligung eines Datenregisters
ANDI	.L .W .B	#Konstante, <ea>			N	Z	0	0	Logisches UND mit einer Konstanten (Direktwert)
EOR	.L .W .B	<ea>, Dn Dn, <ea>			N	Z	0	0	Logisches exklusives ODER unter Beteiligung eines Datenregisters
EORI	.L .W .B	#Konstante, <ea>			N	Z	0	0	Logisches exklusives ODER mit einer Konstanten (Direktwert)
NOT	.L .W .B	<ea>			N	Z	0	0	Bildung des logischen Komplements an effektiver Adresse (1er-Komplement, Umkehrung aller Bits)
OR	.L .W .B	<ea>, Dn Dn, <ea>			N	Z	0	0	Logisches ODER unter Beteiligung eines Datenregisters
ORI	.L .W .B	#Konstante, <ea>			N	Z	0	0	Logisches ODER mit einer Konstanten (Direktwert)

Kapitel 15

Vergleiche

Die Mehrzahl der Befehle setzt Flags im Bedingungscode-Register. Zumeist erhalten wir automatisch Information, ob ein Ergebnis Null war oder nicht (Z-Flag) oder ob es positiv oder negativ in der gewählten Operandenbreite war (N-Flag). Wir können dann durch Auswerten dieser Flags Entscheidungen unmittelbar anknüpfen. Vergleichsbefehle dienen daher der Prüfung komplexerer Muster, auch der Filterung von Wertebereichen, wobei sie durch Setzen der Flags im Bedingungscode-Register die Fragen: Ist ein Wert identisch, kleiner oder größer als ein Vergleichswert? beantworten.

Die CPU führt bei Vergleichen eine gedachte Subtraktion 'Zieloperand minus Quelloperand' aus, ohne einen von diesen zu verändern, setzt aber mit dem Ergebnis die Flags. Man kann nun schnell ablesen: Wenn das Z-Flag = 1 ist, dann sind beide Operanden gleich, außerdem ist das Carry-Flag = 0, weil kein Borgen stattfand. Wenn die Operanden ungleich sind, dann ergibt sich Z = 0. Beim Vergleich vorzeichenfreier Zahlen zeigt C = 1 (Borgen), daß der Zieloperand ungleich und kleiner ist. Und umgekehrt: Z = 0 UND C = 0 heißt: Ungleich, aber größer. Die Verzweigungsbefehle und die Befehlsgruppe DBcc (für Schleifenkonstrukte), die typisch auf einen Vergleichsbefehl folgen, fragen auch diese etwas zusammengesetzten Bedingungen ab.

TST bewirkt eine einfache Operandenprüfung, ob Null oder "negativ" in Bit 31, 15 oder 7 (Operationsbreite) vorliegen.

CHK zur Prüfung von Bereichsüberschreitungen eines Datenregisters sollte man nur benutzen, wenn andernfalls schwerwiegende Folgen zu befürchten sind. Bereichsverletzungen kann man auch mit einfachen Vergleichsbefehlen feststellen, die man im Vordergrund des Programmes mit einer Sperre und einem Bedienerhinweis erledigt.

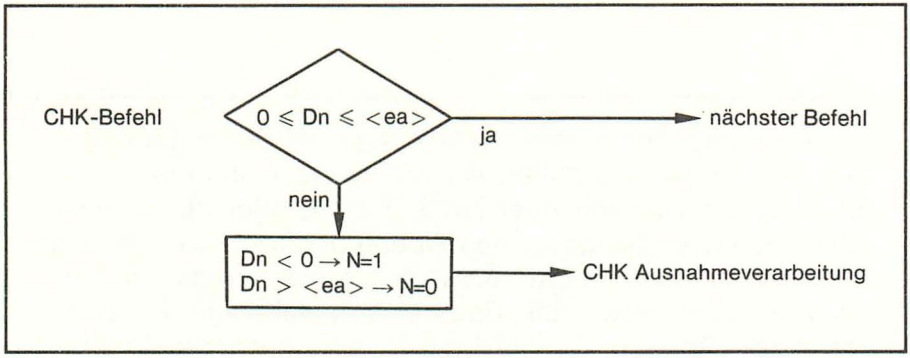


Abbildung 9

Tabelle 9: Vergleiche

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
CHK	(16 Bit)	<ea>, Dn			N	?	?	?	Prüfung eines Datenregisters (untere 16 Bit) auf Bereichsüberschreitung, die zur Ausnahmeverarbeitung Nr. 6 führt, sonst Folgebefehl. Exception mit N = 1, wenn negativ, oder N = 0, wenn Datenregister größer Operand bei <ea>
CMP	.L .W .B	<ea>, Dn			N	Z	V	C	Vergleich mit entsprechendem Setzen der Flags (gedachte Subtraktion: Datenregister – Operand bei <ea>).
CMPI	.L .W .B	#Konstante, <ea>			N	Z	V	C	Vergleich wie CMP (gedachte Subtraktion: Zieloperand – Direktwert). Der Zieloperand wird nicht verändert.
CMPM	.L .W .B	(Ay)+, (Ax)+			N	Z	V	C	Vergleich von Speicherzellenbereichen zu aufsteigenden Adressen (gedachte Subtraktion: Zieloperand – Quelloperand) und Setzen der Flags
TST	.L .W .B	<ea>			N	Z	0	0	Prüfung des Operanden an effektiver Adresse auf Vorzeichen (höchstwertiges Bit) und Null

Kapitel 16

Befehle zur Bitmanipulation und Bitprüfung

Für das Prüfen und Ändern einzelner Bits kann ein Muster in einem Datenregister oder in einem Direktwert benutzt werden. Wenn der Zieloperand ein Datenregister ist, dann kann jedes seiner 32 Bits mit diesen Befehlen in Bearbeitung genommen werden. Steht der Zieloperand im Speicher, so arbeiten die Befehle byte-weise. Zieloperanden können nicht die Adreßregister sein (wozu auch?) und nicht Zellen, die relativ zum Programmzähler adressiert werden sollen. Alle Datenregister und über Adreßregister indirekt angesprochene Ziele sind erreichbar.

Der Befehl TAS hat eine besondere Rolle: Seine Zyklen des Lesens/Bearbeitens/Zurückschreibens sind unteilbar, können also nicht unterbrochen werden (Exception-Ereignisse). Diese Einrichtung wurde geschaffen, damit verschiedene Prozesse auf gleiche Einrichtungen (Datenbanksätze oder Peripherie) Zugriff nehmen können, ohne sich zu stören. Ein Programm untergeordneter Priorität ist in der Lage, ein Flag per TAS ungestört zu setzen. Ein anderer per Interrupt während des TAS-Befehles hereinbrechender Prozeß stößt dann auf ein noch besetztes Gerät oder auf ein Datenfeld, das sich noch in Aufarbeitung befindet und bemerkt, daß er hier noch nicht zugreifen darf.

Tabelle 10: Bitmanipulation, Bitprüfung

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
BCHG	(32/8 Bit)	Dn, <ea> #Konstante, <ea>				Z			Prüfe ein spezifiziertes Bit im Zieloperanden und ändere (invertiere) es.
BCLR	(32/8 Bit)	Dn, <ea> #Konstante, <ea>				Z			Prüfe ein spezifiziertes Bit im Zieloperanden und setze es auf Null.
BSET	(32/8 Bit)	Dn, <ea> #Konstante, <ea>				Z			Prüfe ein spezifiziertes Bit im Zieloperanden und setze es auf Eins.
BTST *)	(32/8 Bit)	Dn, <ea> #Konstante, <ea>				Z			Prüfe, ob ein einzelnes Bit im Zieloperanden gesetzt ist.
TAS	(8 Bit)	<ea>			N	Z	0	0	Unteilbarer Befehl (Zugriffsschutz). Prüfe Bit 7 im Zieloperanden (Byte) und kopiere es in das N-Flag, setze Bit 7 im Zieloperanden zu "1". Das Z-Flag entspricht dem Inhalt des Byte-Operanden (ob Null oder ungleich Null).

ANMERKUNG: *) Im Status wird das Condition Code Register im Z-Bit zu "1" gesetzt, wenn das spezifizierte Bit des Zieloperanden vor der Operation Null war, sonst Z = 0.

Zieloperand kann sein
a) ein Datenregister, dann sind alle 32 Bits ansprechbar, Langwortverarbeitung
b) eine Zelle im Speicher, dann arbeitet der Befehl auf der Basis Byte (8 Bit)

Die Numerierung eines Bits erfolgt
a) Modulo 32 bzw.
b) Modulo 8

Kapitel 17

Schiebe- und Rotierbefehle

Diese Befehlsgruppe wird typisch bei Multiplikationen und Divisionen mit erhöhter Genauigkeit benutzt, wenn Datenfelder der Operanden gegeneinander verschoben werden müssen. Was als Übertrag aus dem einen Feld herauskommt, kann in das nächste Feld eingezogen werden. Eine Linksverschiebung um jeweils eine Stelle bedeutet binär eine Multiplikation um eine Zweierpotenz (es gibt ja auch den Multiplikationsbefehl) und umgekehrt für die Division. Bei der Bearbeitung von BCD-Zahlen verschiebt man um jeweils 4 Positionen (eine Zehnerpotenz).

Einzelne Bits sollte man mit dieser Befehlsgruppe nicht untersuchen oder verändern wollen. Dafür gibt es die Bit-Befehle und die Gruppe der logischen Befehle. Sie werden jedoch sinnvoll angewandt, wo man Operanden schrittweise in ein anderes Darstellungsformat umwandeln möchte, z. B. ASCII-Zeichen in Binär-oder BCD-Zahlen und umgekehrt.

Die Befehlswirkung ist beim Erzeugen eines Lauflichtes an Interfacebausteinen gut beobachtbar, muß aber in dieser Form nicht nachvollzogen werden.

Bei der arithmetischen Rechtsverschiebung mit dem Befehl ASR ist zu beachten, daß das höchstwertige Bit im Operanden immer erhalten bleibt, da es ein Vorzeichen darstellt. Mit jedem Verschiebeschritt dehnt sich dieses Vorzeichen nach recht in die Operandenstellen aus.

Tabelle 11: Schiebe- und Rotierbefehle

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung X	N	Z	V	C	Befehlswirkung
ASL LSL	.L .W .B	Dx, Dy #data, Dy <ea>	C	N	Z	V	C	Arithmetische und logische Linksverschiebung. Von rechts werden Nullen nachgezogen. Das herausgeshiftete Bit geht zugleich ins Carry- und Extendbit.
ASR	.L .W .B	Dx, Dy #data, Dy <ea>	C	N	Z	V	C	Arithmetische Rechtsverschiebung. Das Vorzeichenbit wird ins sich selbst kopiert. Das herausgeshiftete Bit geht zugleich ins Carry- und Extendbit.
LSR	.L .W .B	Dx, Dy #data, Dy <ea>	C	N	Z	V	C	Logische Rechtsverschiebung. Von links werden Nullen nachgezogen. Das herausgeshiftete Bit geht zugleich ins Carry- und Extendbit.
ROL	.L .W .B	Dx, Dy #data, Dy <ea>		N	Z	V	C	Rotieren nach links. Das vorderste Bit geht zugleich ins Carry und in die niedrigste Bitposition.
ROR	.L .W .B	Dx, Dy #data, Dy <ea>		N	Z	V	C	Rotieren nach rechts. Das niederwertigste Bit geht zugleich in das Carry und in die höchste Bitposition.
ROXL	.L .W .B	Dx, Dy #data, Dy <ea>	C	N	Z	V	C	Rotieren nach links mit Extendbit (X) aus Vortrag: Es geht in die niedrigste Bitposition. Das höchstwertige Bit geht zugleich ins Carry- und Extendbit.
ROXR	.L .W .B	Dx, Dy #data, Dy <ea>	C	N	Z	V	C	Rotieren nach rechts mit Extendbit (X) aus Vortrag: Es geht in die höchste Bitposition. Das herausgeshiftete Bit geht ins Carry- und Extendbit.

ANMERKUNGEN: a) Wenn der Operand im Speicher steht, dann Operation immer in Wortbreite und Verschiebung nur um eine Stelle!
b) Direktwerte (data) nur in Wertebereich 1–8 für Verschiebungen in Datenregistern.
c) Bei Zahl der Verschiebeschritte im Register Dx Bewertung Modulo 64
d) Bei ASL, LSL, ASR und LSR wird das Flag V=1, sobald sich das höchstwertige Bit überhaupt einmal änderte, sonst: V=0.

Kapitel 18

Sprünge, Unterprogramme, numerierte Traps

Die Befehle **BRA** (Verzweige immer) und **JMP** (unbedingter Sprung) sind in ihrer Wirkung gleichwertig: Der Programmzähler wird mit einem neuen Wert geladen, der sich aus dem Offset in BRA oder aus der Adresse im JMP-Befehl ergibt. Letztere wird erst vom Linker endgültig berechnet. Man kann aber empfehlen, den BRA-Befehl möglichst häufig im "Nahfeld" zu benutzen, wo das Verzweigungsziel nicht allzu weit entfernt ist. Der Jump-Befehl läßt sich aber wegen seiner vielen Adressierungsarten für Fallentscheidungen ähnlich einem berechneten GOTO benutzen.

Das gilt auch für die sonst gleichwertigen Befehle **BSR** (Verzweigung zu einem Unterprogramm) und **JSR** (Jump to Subroutine). Am Ende solcher Unterprogramme steht normalerweise der Befehl **RTS** für die unbedingte Rückkehr.

Bei der Anlage von Unterprogrammen sollte man auch den Befehl **LINK** in Betracht ziehen, mit dem man ihnen einen geschützten Parameterbereich (lokal und vorübergehend) auf dem Stack verschaffen kann. Mit dieser Einrichtung ist es dann auch möglich, daß ein Unterprogramm sich selbst rekursiv aufrufen kann, ohne daß die Parameter der höheren Aufrufebenen gestört werden. Oder: Wo mehrere Prozesse, vielleicht sogar im Interruptbetrieb, die gleiche Dienstleistung fast zugleich und überlappend benutzen, da sollte man solche geschützten Parameterbereiche mit LINK einrichten.

Die 16 numerierten Traps **TRAP#0 ... TRAP#15** sind eine besondere Einrichtung, auf die schon bei den Exceptions (Ausnahmeverarbeitung) hingewiesen wurde; sie lösen Unterbrechungen des Programmablaufes per Befehl aus (Software-Interrupte). Die Zentraleinheit wird gezwungen, einen Langwort-Vektor aus dem Adressenbereich $80 ... $BF in den Programmzähler zu laden, was die Programmfortsetzung an der vektorierten Adresse nach sich zieht. Diese 16 Vektoren sind natürlich vom Betriebssystem bei der Initialisierung vorbesetzt worden, könnten jedoch auch vom Anwender auf seine Routinen "umgebogen" werden.

TRAP-Befehle	Vektoradresse:	Befehl:	Vektornummer:
	$80 - $83	TRAP #0	$20
	$84 - $87	TRAP #1	$21
	$88 - $8B	TRAP #2	$22
	$8C - $8F	TRAP #3	$23
	$90 - $93	TRAP #4	$24
	$94 - $97	TRAP #5	$25
	$98 - $9B	TRAP #6	$26
	$9C - $9F	TRAP #7	$27
	$A0 - $A3	TRAP #8	$28
	$A4 - $A7	TRAP #9	$29
	$A8 - $AB	TRAP #10	$2A
	$AC - $AF	TRAP #11	$2B
	$B0 - $B3	TRAP #12	$2C
	$B4 - $B7	TRAP #13	$2D
	$B8 - $BB	TRAP #14	$2E
	$BC - $BF	TRAP #15	$2F

Abbildung 10: TRAP-Befehle

Es sind diese numerierten Traps, die beim Eindringen des Programmierers in das Betriebssystem eine zentrale Bedeutung haben. Von Atari sind benutzt:

TRAP #1 für GEMDOS-Aufrufe (zeigt auf Adresse $965E im Betriebssystem, das in das RAM geladen wird)
TRAP #2 für GEM-Dienste ($2A338)
TRAP #13 für BIOS-Aufrufe (Basic Input Output System)
TRAP #14 für XBIOS (erweitertes BIOS).

Wenn auch die Benutzung der Traps mehr Zyklen in der Ausführung benötigt, als ein einfacher symbolischer Aufruf z. B. mit JSR AUSGABE, so hat das TRAP-System doch seine Vorteile: Betriebssystem-Adressen können sich von Version zu Version verändern, was Verwirrung hervorrufen wird. Mit den Traps und einer gleichbleibenden Syntax für ihre Benutzung erreicht man nun unabhängig von Adressen gleiche Leistungen aus dem Betriebsprogramm. Dem Systemprogrammierer kann es daher gleichgültig sein, ob das Betriebssystem im RAM oder im Festwertspeicher liegt, wenn er die Traps benutzt.

Tabelle 12: Sprünge, Unterprogramme, numerierte Traps

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
BRA	.W .B	<Label>							Unbedingte Programmverzweigung (Branch Always) kurz (+127 bis –128) oder lang (+32767 bis –32768)
BSR	.W .B	<Label>							Unbedingte Verzweigung zu einem Unterprogramm, von dem man mit RTS auf den Folgebefehl zurückkehrt. Verzweigungsweiten wie bei BRA.
JMP		<ea>							Unbedingte Programmfortsetzung an anderer Stelle. Effektive Adressen absolut kurz und lang, Adreßregister indirekt, auch mit Displacement und Index, ferner PC-relativ. d(PC,Rx)
JSR		<ea>							Unterprogrammaufruf mit den gleichen vielfältigen Adressierungsarten
RTE			S	X	N	Z	V	C	Privilegierter Befehl! Rückkehr von einer Ausnahmeverarbeitung (Exception). Der Programmzähler und das gesamte Statuswort werden vom Stack zurückgeladen.
RTR				X	N	Z	V	C	Rücksprung mit Laden der Flags, allerdings ohne Veränderung der Trace- und Supervisorbits oder der Interruptmaske.
RTS									Normale Rückkehr aus einem Unterprogramm.
TRAP #		# Vektor-Nummer							Unbedingtes numeriertes Trap: Programmzähler und Statuswort auf dem Stack retten, numerierten Trapvektor in den PC laden. Programmfortsetzung an der geladenen Adresse.

Kapitel 19

Bedingte Verzweigungen, Schleifenkonstrukte

Verzweigungen in Abhängigkeit von eingetretenen Bedingungen (Bedingungscode-Register im Status) gehören zu den am häufigsten benutzten Befehlen. Zu den einfachen Abfragen gehört dabei, ob ein Operand Null oder ungleich Null war (BEQ, BNE) oder ob positiv oder negativ (Entscheidungspaar BPL, BMI). Abfragen dieser Art knüpfen daher typisch an vorausgegangene Befehie wie MOVE, CMP, TST oder Bitbefehle an. Die etwas weitergreifenden Entscheidungen dienen dem Filtern von Wertebereichen und dem Rechnen im Zweierkomplement.

Schleifenkonstrukte: Zählschleifen werden in fast jedem Programm benutzt. Für sie gibt es zusammengesetzte Befehle in der Gruppe DBcc: Prüfe Bedingung, dekrementiere und verzweige (Test Condition, Decrement and Branch). Die Abläufe sind hier wie folgt: Zuerst wird die Bedingung im CCR geprüft. Trifft sie zu, so wird der Inhalt des Datenregisters Dn um 1 dekrementiert. Wenn die Zählung nun abgelaufen ist (Das Wort in Dn ist $FFFF geworden), dann wird ebenfalls der Folgebefehl ausgeführt. Andernfalls (Bedingung nicht erfüllt, Zähler nicht abgelaufen) wird in die Schleife zurückverzweigt. Die Befehle arbeiten in Wortbreite, und zwar sowohl für die Vorladung des Zählers in Dn wie auch für den Offset in der Verzweigungsweite.

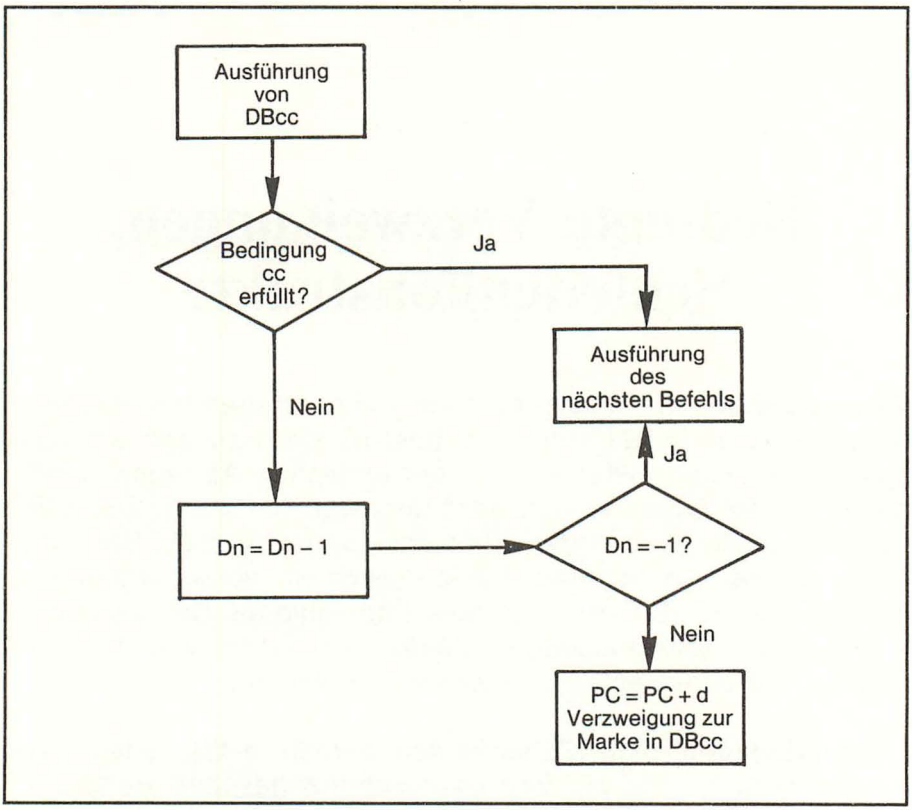


Abbildung 11: Abläufe beim Befehl DBcc

TRAPV ist ebenfalls eine bedinge Programmfortsetzung an anderer Stelle (per TRAPV-Vektor), wenn ein Überlauf eingetreten ist.

Tabelle 14: Bedingte Verzweigungen, TRAPV

Befehl	Operanden	Syntax, Quelle, Ziel	Statusabhängigkeit						Befehlswirkung: Verzweige, wenn ...
				X	N	Z	V	C	
BCC BCS		<Label>						C	Carry Clear (= 0) Carry Set (= 1)
BEQ BNE		<Label>				Z			Gleich (Equal) Ungleich (Not Equal)
BHI BLS		<Label>				Z		C	Größer (Higher) Kleiner oder gleich (Less or Same)
BPL BMI		<Label>			N				Positiv (N = 0) Negativ (N = 1)
BVC *) BVS *)		<Label>					V		V-Flag gelöscht V-Flag gesetzt
BGE *) BLT *)		<Label>			N		V		Größer oder gleich (Greater or Equal) Kleiner als (Less Then)
BGT *) BLE *)		<Label>			N	Z	V		Größer (Greater Then) Kleiner oder gleich (Less or Equal)
TRAPV *)		–					V		Programmfortsetzung mit TRAPV-Vektor, wenn V-Flag gesetzt (Overflow), sonst Folgebefehl nehmen.

ANMERKUNG: Mit *) sind Befehle gekennzeichnet, die mit Zweierkomplement-Arithmetik arbeiten..

Tabelle 13: Schleifenkonstrukte

Befehl	Operanden	Syntax, Quelle, Ziel	Statusabhängigkeit						Befehlswirkung: Verlasse, wenn ... oder wenn Dn negativ geworden ist
				X	N	Z	V	C	
DBCC DBCS	.W	Dn, Label						C	Carry Clear Carry Set
DBEQ DBNE	.W	Dn, Label				Z			Gleich Ungleich
DBHI DBLS	.W	Dn, Label				Z		C	Größer Kleiner oder gleich
DBPL DBMI	.W	Dn, Label			N				Positiv Negativ
DBVC *) DBVS *)	.W	Dn, Label					V		V-Flag gelöscht V-Flag gesetzt
DBGE *) DBLT *)	.W	Dn, Label			N		V		Größer oder gleich Kleiner als
DBGT *) DBLE *)	.W	Dn, Label			N	Z	V		Größer Kleiner oder gleich
DBT DBF	.W	Dn, Label							Immer (True) Niemals (False)

ANMERKUNG: Mit *) bezeichnete Befehle arbeiten mit der Arithmetik für Zweierkomplement zusammen.
Grundsätzlich Wortbreite für die Zählvorgabe in Dn und die Verzweigungsweite.
Wenn beim DBxx-Befehl die Bedingung zutrifft, wird der Folgebefehl ausgeführt, ebenso, wenn Dn –1 geworden ist.

Kapitel 20

Stackverwaltung, verschiedene Befehle

Das Befehlspaar **LINK** und **UNLK** dient der Anlage und späteren Freigabe geschützter Arbeitsbereiche auf dem Stack. Es wurde speziell für die Implementierung höherer Sprachen geschaffen, die mit "lokalen" Parametern arbeiten, und für Umgebungen, in denen mehrere Tasks oder Benutzer in Konkurrenz Zugriff auf die Ressourcen des Computers nehmen.

Zum Ablauf bei LINK das beifolgende Schema nebst Erklärungen: Nach dem Eintritt in ein Unterprogramm wird ein Adreßregister An zunächst einmal auf den Stack per Adressierungsart –(A7) gerettet, damit es den jetzigen Inhalt des Stackpointers aufnehmen kann. Dieser wird automatisch nach An übertragen. Sodann wird der Offset im Befehl auf den Stackpointer aufaddiert. Eine sinnvolle Speicherreservierung erreicht man nur mit negativem Offset, weil ein Stackpointer sich zu niedrigen Adressen hin bewegen soll (bei Beschickungen).

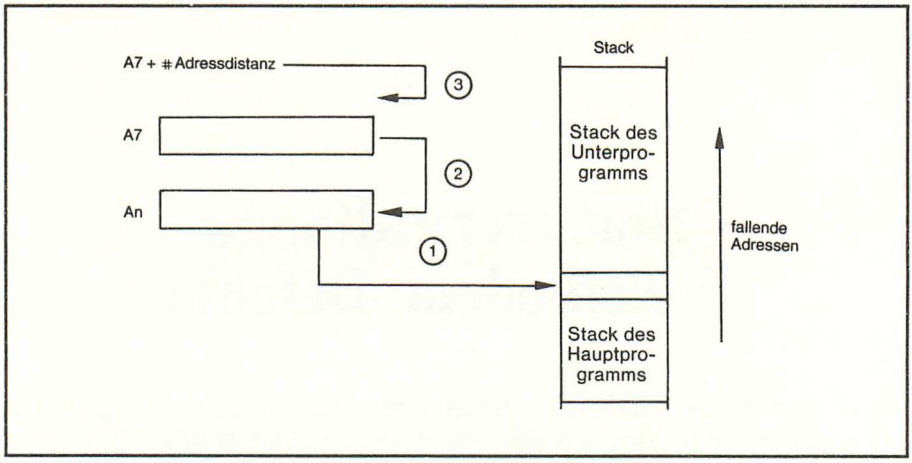


Abbildung 12: LINK rettet An und versetzt A7

Der per Versetzen des Stackpointers geschaffene Zwischenraum ist jetzt der geschützte Parameterbereich des Unterprogrammes, in den man gewöhnlich Werte transportiert. – Parameterübergaben vom und an das rufende Programm können jetzt leicht erfolgen. Das Adreßregister An steht ja in einem definierten Abstand zu Parametern, die der Rufer ggfs. auf den Stack gebracht hat. Parameter können dabei auch Zeiger sein, die man mit PEA auf den Stack brachte.

Der Befehl UNLK bringt den Stackpointer und das Adreßregister An automatisch auf die Werte zurück, die vor dem Aufruf mit LINK bestanden.

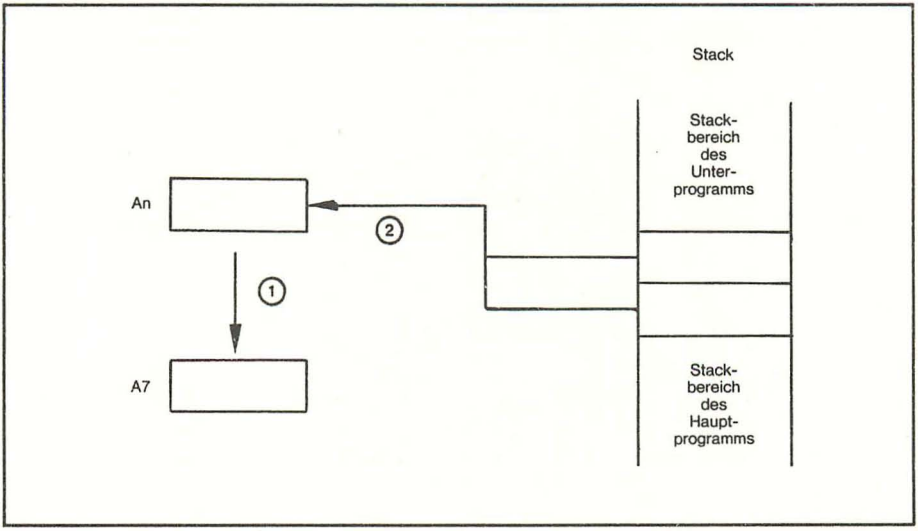


Abbildung 13: UNLK holt A7 und An zurück

Tabelle 16: Stackverwaltung, Parameterübergabe

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung X	N	Z	V	C	Befehlswirkung
LINK	.L .W*	An, # <offset>						Reservieren eines Stackbereiches (Frame). Der Inhalt des Adreßregisters An wird auf den Stack gerettet. Der berichtigte Stackpointer wird nach An kopiert (für den Befehl UNLK). Der Stackpointer wird dann um den Offset verändert. Ein negativer Offset reserviert einen Stackbereich. Ein Offset als Wortoperand wird vorzeichenerweitert verwertet.
UNLK		An						Freigabe eines Stackbereiches nach früherem LINK: An in den Stackpointer kopieren, ein Langwort (das frühere An) vom Stack ziehen und nach An laden und den SP um 4 erhöhen.
PEA	(32 Bit)	<ea>						Ablage einer berechneten effektiven Adresse auf dem aktiven Stack (User- oder Systemstack).
MOVE USP	(32 Bit)	USP, An An, USP						Privilegierter Befehl! Im Supervisor-Modus Laden eines Adreßregisters aus dem Stackpointer oder umgekehrt. Im User-(Anwender-)Modus: Exception wegen Privilegverletzung.

ANMERKUNG: W* bedeutet Vorzeichenerweiterung auf 32 Bit.

Tabelle 15: Verschiedene Befehle

Befehl	Operanden	Syntax, Quelle, Ziel	Statusbeeinflussung						Befehlswirkung
				X	N	Z	V	C	
NOP									No Operation = Keinerlei Befehlsausführung. Beendigung aller Buszyklen, zur Synchronisation geeignet.
RESET									Privilegierter Befehl! Im Supervisor-Modus Ausgabe von elektrisch Low am Reset-Pin an die Peripherie (124 Zyklen). Im User-Modus: Exception Nr. 8.
STOP	(16 Bit)	#data	alle 16 Bits						Privilegierter Befehl! Im Supervisor-Modus Setzen des Statusregisters gemäß Direktoperand und Anhalten des Prozessors, Programmfortsetzung, wenn Trace zuvor aktiv oder Interrupt gemäß Priorität akzeptierbar. Im User-Modus: Privilegverletzung (Exception).

Kapitel 21

Adressierungsarten
– die effektive Adresse –

Der 68000 hat viele Adressierungsarten und auch verschiedene Möglichkeiten, Befehle auf die Breite der zu bearbeitenden Operanden einzustellen. Der programmierende Einsteiger mag dadurch zunächst etwas ratlos sein, wie er das alles nutzen soll. Die Prinzipien sind dennoch einfach, sie müssen nur erklärt sein. – Die zentrale Frage ist, wie der 68000 die sogenannten effektive Adresse zur Ansprache von Operanden bildet.

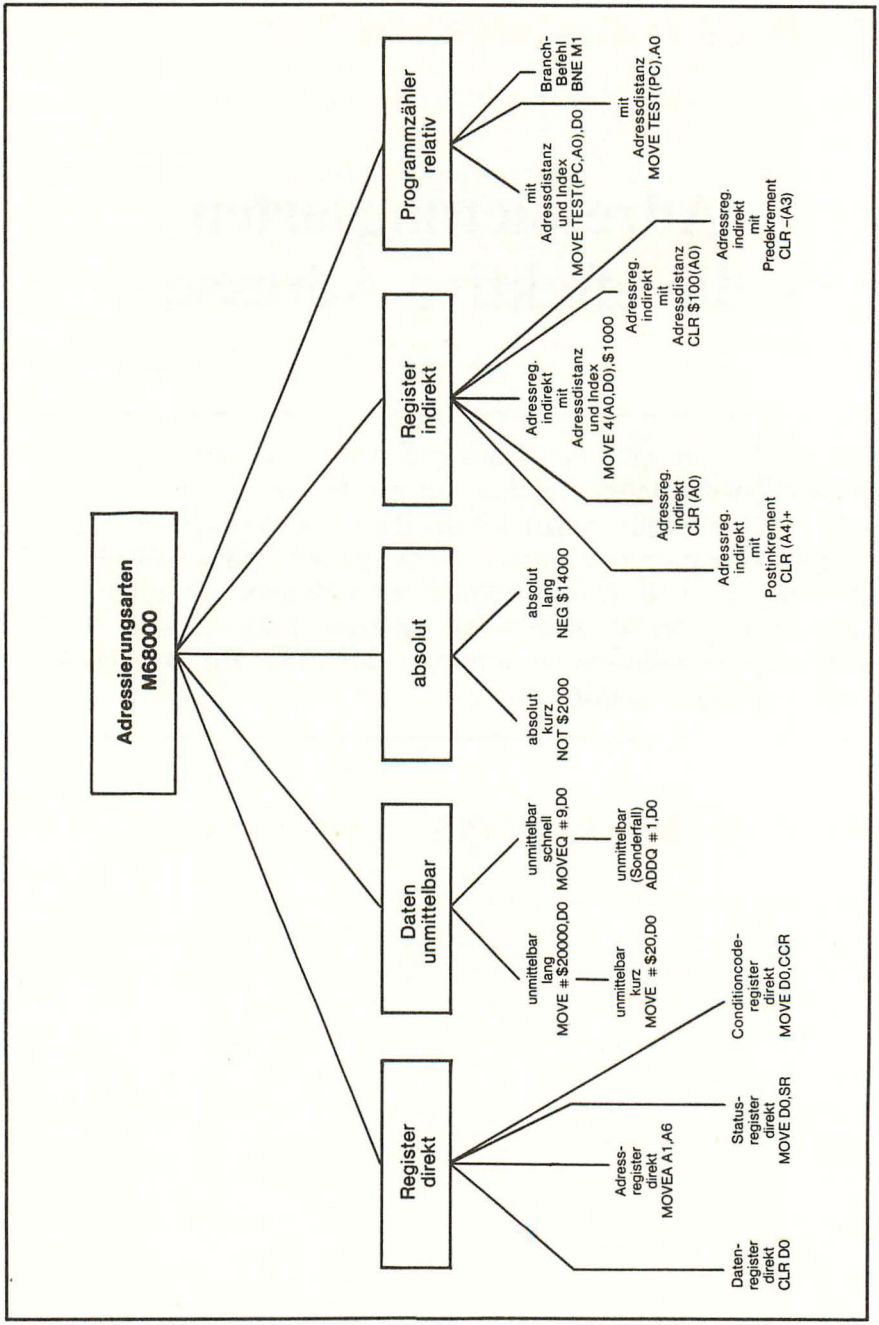


Abbildung 14: Adressierungsarten – Übersicht

CPU-Register als Quelle oder Ziel

Die ein oder zwei Operanden, die ein Befehl ansprechen soll, können sich in den Registern der CPU oder im Speicher befinden. Wenn ein CPU-Register Quelle oder Ziel ist, so sind die Verhältnisse für den Leser einfach zu überschauen: Das interne Schaltwerk öffnet die Verbindungswege für Transporte und Bearbeitungen, es muß keine Adresse an den Adreßbuspins ausgesandt werden. Registerbezeichnungen, die in das Befehlswort hineincodiert sind, implizieren bereits die effektiven Adressen.

Direktoperanden

Einfach ist es auch noch, wenn ein Direktoperand (Konstante) im Befehl benutzt wird. Er steht dann im Wort bzw. in den Worten des Befehles im Programmspeicher. Die Adressierung ist mit der Benutzung des Operators "#" (für Direktwert) implizit erledigt. Das gilt auch für Befehle wie TRAP #.

Befehle mit feststehenden Adressierungen

Bekannt ist auch, daß einige Befehle keinerlei Adressierung benötigen, wie z. B. NOP (No Operation) oder RTS (Rückkehr aus einem Unterprogramm). Bei letzterem wird nach feststehenden Regeln eine Rückkehradresse aus dem Stapelzeiger in den Programmzähler geladen. Dem Leser werden sogleich auch die Verzweigungsbefehle einfallen, die den Programmzähler um einen Offset von +–7 oder +–15 Bit verändern, wenn die Verzweigungsbedingung zutrifft.

Adressierung des Speichers

Wir müssen bezüglich der effektiven Adresse jetzt nur noch Adressierungen genauer erklären, mit denen der Speicher angesprochen wird.

Absolute Adressierung

Dem Leser dürften die absoluten Adressierungen des Speichers ausreichend bekannt sein. Sie lesen sich etwa JMP $E000, Springe zur Adresse $E000, oder MOVE.W $1000,$2000, übertrage ein Wort aus den Adressen $1000/1001 nach $2000/2001. In solchen Fällen steht die effektive Adresse in den Befehlsworten, wo man sie mit dem Disassembler ablesen kann.

Die Bildung anderer effektiver Adressen ist damit auf Fälle reduziert, in denen der Inhalt eines Adreßregisters oder des Programmzählers herangezogen wird.

Ein Adreßregister als Zeiger

Die CPU ist so konzipiert, daß ihre 8 Adreßregister als Zeiger auf den Speicher benutzt werden können. Bekanntlich sind sie 32 Bit breit und können damit eine volle Adresse aufnehmen, die so wie sie ist, auf den Adreßbus ausgesandt oder zunächst noch als Rechenbasis benutzt wird. In jedem Falle sind wir in der Gruppe der 'indirekten Adressierungen über ein Adreßregister'. Hier die Bildungsregeln für effektive Adressen und ihre Schreibweisen:

(An) **Adreßregister indirekt:**
Was in An steht, ist die auszusendende effektive Adresse

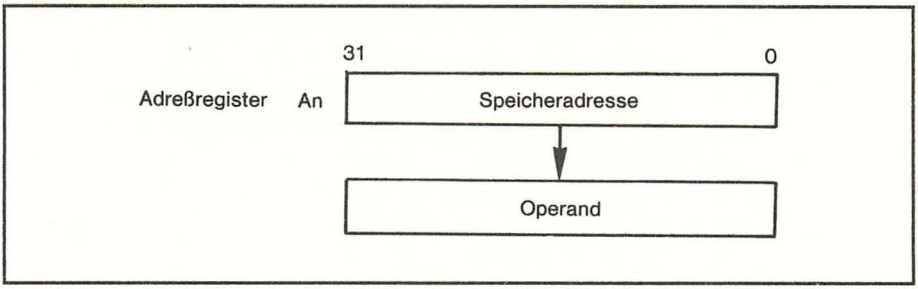


Abbildung 15: Effektive Adresse im Adreßregister

d(An) **Adreßregister indirekt mit Adreßdistanz:**
d ist eine vorzeichenbehaftete Adreßdistanz von 15 Bit. Die effektive Adresse verändert sich damit um bis zu +–32768 gegenüber dem Inhalt von An. An wird als Zeiger auf einen Bereich benutzt.

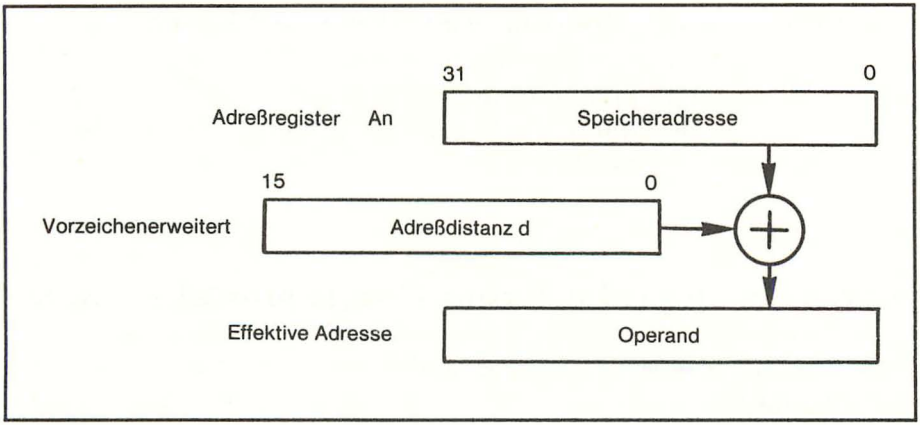


Abbildung 16: Offset zum Inhalt des Adreßregisters

d(An,Rx) **Adreßregister indirekt mit Index und mit/ohne Adreßdistanz:**
d ist hier eine vorzeichenbehaftete Adreßdistanz von +–7 Bit. Rx ist irgendein Daten- oder Adreßregister der CPU, dessen Inhalt vorzeichenerweitert als Indexregister zu (An) und d hinzuaddiert wird. Da im Indexregister zuvor Berechnungen durchgeführt werden können, ist mit dieser Adressierungsart ein berechneter Zugriff auf Datenelemente möglich, die sich relativ zu (An) befinden.

–(An) **Adreßregister indirekt mit Pre-Dekrement:**
Je nach eingestellter Operandenbreite für den Befehl, also z. B. MOVE.L, MOVE.W oder MOVE.B, wird der Inhalt des Registers An **vor** der Befehlsausführung zunächst um 4, 2 oder 1 erniedrigt. Hier ist eine Automatik eingebaut, die bei Stackbeschickungen üblich ist: Bearbeitung von Datenfeldern zu absteigenden Adressen hin.

(An)+ **Adreßregister indirekt mit Post-Inkrement:**
Analog zur Adressierung –(An), jedoch Arbeiten zu aufsteigenden Adressen hin. Erhöhung von An **nach** der Befehlsausführung um die Operandenbreite.

Adressierung relativ zum Programmzählerstand

Oft sollen Programme an jede Stelle des Speichers ladbar und dort ausführbar, also voll verschieblich sein. Sie dürfen dann keine absoluten Adressen enthalten, die Referenzen auf das Programm selbst bilden. In gewissen Grenzen erreicht man das bei Verwendung von Branch-Befehlen, wo man sonst auch JMP oder JSR schreiben darf. Branches arbeiten mit einem Offset zum Programmzähler. Dieser Grundgedanke gilt auch für die

Adressierungen relativ zum Programmzähler, mit denen nun auch Operandenansprachen, Sprünge usw. von der Ladeadresse unabhängig werden. Als Programmierer brauchen wir uns nicht um die Berechnungen während der Assemblierung und während der Ausführung zu kümmern, wenn wir eine der Schreibweisen d(PC) oder d(PC,Rx) benutzen. Klar ist, daß jeder Operand eines Programm-Modules einen Offset zum Programmzählerstand in dem Moment hat, wo der Operand per Befehl angsprochen wird. Diese relative Entfernung wird in den Befehl hineincodiert. Im einzelnen:

d(PC) **Programmzähler-relative Adressierung mit/ohne Adreßdistanz:**
d ist eine vorzeichenbehaftete Adreßdistanz von 15 Bit. Die effektive Adresse verändert sich damit um bis zu +–32768 gegenüber dem Inhalt des Programmzählers, der als Adreßbasis dient.

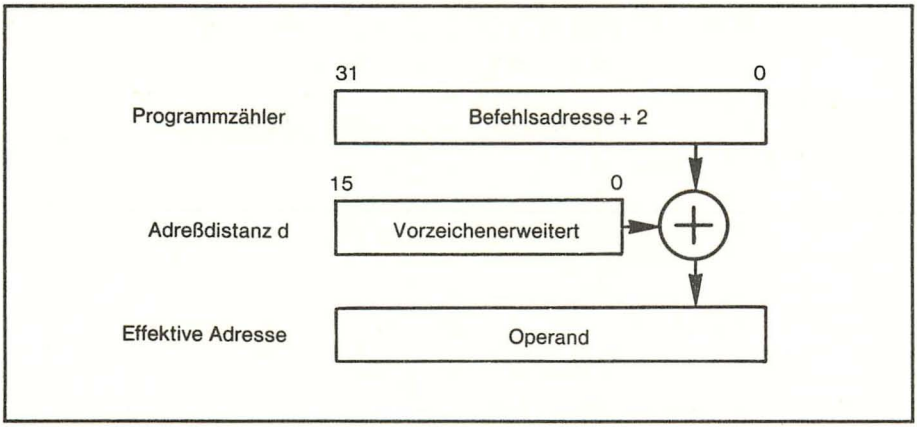


Abbildung 17: Der Programmzähler als Adreßbasis

d(PC,Rx) **Programmzähler-relativ mit Index und mit/ohne Adreßdistanz:**
d ist hier eine vorzeichenbehaftete Adreßdistanz von +–7 Bit. Rx ist irgendein Daten- oder Adreßregister der CPU, dessen Inhalt vorzeichenerweitert als Indexregister zu (PC) und d hinzuad-

diert wird. Da im Indexregister zuvor Berechnungen durchgeführt werden können, ist mit dieser Adressierungsart ein berechneter Zugriff auf Datenelemente möglich, die sich relativ zum Programmzähler befinden.

Das nachfolgende Beispiel macht die unterschiedlichen Adressierungen deutlich: In Zeile 2 erfolgt ein absoluter Sprung nach LABEL, in Zeile 3 ein PC-relativer. Wir sehen, daß sein Code eine Adreßdistanz von 6 annimmt, die sich aus $E (LABEL) minus $8 ergibt. $8 ist das Wort hinter dem eigentlichen Befehlsbyte von JMP und ist Berechnungsbasis – wie z. B. bei Verzweigungen.

Text

```
 1 00000000                    TEXT
 2 00000000 4EF90000000E        JMP LABEL ABSOLUTER SPRUNG
 3 00000006 4EFA0006            JMP LABEL(PC) SPRUNG RELATIV ZUM PC
 4 0000000A 4E71                NOP
 5 0000000C 4E71                NOP
 6 0000000E 4E71               LABEL NOP SPRUNGZIEL
 7
 8 00000010                    .END

S y m b o l   T a b l e
LABEL     0000000E TEXT
```

Kapitel 22

Die wichtigen Direktiven

Anweisungen an das Assemblerprogramm, Direktiven, werden auch Pseudo-Opcodes genannt. Sie sind Formulierungen im Quelltext. Die meisten wirken sich während des Assembler-Laufes aus, einige wirken bis hin in die Stufe des Linkers. In diesem Abschnitt besprechen wir die zum täglichen Gebrauch gehörenden Anweisungen. Feinheiten werden bei "Sonstigen Direktiven" besprochen.

Man unterscheide die im Quelltext enthaltenen Direktiven von Anweisungen, die man beim Hereinrufen des Assembiers (besonderer Abschnitt) in der Kommandozeile erteilt.

Direktiven wie z. B. EQU können wahlweise auch mit vorangestelltem Punkt geschrieben werden.

Bereiche TEXT, DATA und BSS

In einem Programm kann man drei Bereiche nach ihrer Funktion unterscheiden:

Bereich der Befehle. Er wird mit dem reservierten Wort TEXT angesprochen. Befehle unterliegen der Umrechnung durch den Linker auf absolute Adressen. Der Assembler erzeugt für diese Umrechnung in seinem Objectfile Name.0 Zusatzinformationen für Befehle. Daher der besondere TEXT Abschnitt.

Bereich für Tabellen der verschiedensten Art. Sie werden in den besonderen Bereich DATA gebracht. Es handelt sich um initialisierte Daten, die mit dem Laden des Programmes als Konstanten schon im Speicher stehen. Diese Daten werden im Quelltext mit der Direktive .DC Ausdruck erzeugt (Define Constant).

Arbeitsbereiche für Variablen. Es handelt sich um nicht initialisierten Speicherraum im RAM, in den das Programm seine Operanden ablegt. Dieser Bereich wird mit der Direktive .BBS (Block Storage Segment) angesprochen. Speicherraum wird hier mit der Direktive .DS (Define Storage) reserviert, wobei auch hier, wie in den Segmenten TEXT und DATA, Label gebraucht werden dürfen.

Weil die Zusammentragung eines Programmes aus Modulen letztlich dem Linker überlassen bleibt, darf man bei einer Programmierung durchaus zwischen diesen drei Grundsegmenten hin- und herspringen.

Die Direktiven zur Bezeichnung der Segmente sind damit TEXT, DATA und BSS (bzw. auch .TEXT, .DATA und .BSS). Die nachfolgende Liste enthält hierzu ein Beispiel.

Symbolerklärungen mit .EQU oder "=" und SET

EQU ist die Abkürzung für 'Equate Symbol Value', Gleichsetzung eines Symboles mit einem Wert. Die Schreibweise ist **SYMBOL .EQU Ausdruck**. Ein Ausdruck ist dabei ein Wert aus Zahlen und/oder Symbolen und Operatoren, wie wir es im Abschnitt zur typischen Befehlszeile dargestellt haben. Bei jedem Gebrauch eines Symboles setzt der Assembler stattdessen den Wert des zugeordneten Ausdruckes ein. Das Gleichheitszeichen hat die gleiche Wirkung wie .EQU oder EQU. Ein Symbol kann auch mit

SET definiert werden. Sein Wert darf dann im weiteren Verlauf des Textes mit einer weiteren SET-Anweisung verändert werden.

Konstanten definieren mit .DC

DC steht für Define Constant. Die Schreibweise ist: **Label .DC.X Ausdruck**. Mit "X" wird das Speicherformat bestimmt. Also .DC.L, .DC.W oder .DC.B für Langwort-, Wort- und Byte-Konstanten. Von der Wertdarstellung nicht benutzte führende Bitstellen werden dabei ggfs. ausgenullt. Man beachte, daß Zeichenketten ebenfalls mit .DC erklärt werden können, wenn man einfache oder doppelte Anführungsstriche benutzt (Gänsefüßchen). Die Verwendung eines Labels ist natürlich optional. Mit einer .DC-Zeile können mehrere Konstanten erzeugt werden, wenn ihre Argumente mit Komma getrennt sind.

Arbeitsspeicher mit .DS reservieren

Innerhalb des Segmentes .BBS (Block Storage Section) wird Platz mit "Define Storage" reserviert und mit optionalen Labeln versehen. Die Schreibweise ist **Label .DS.X Menge.** steht dabei wieder für L, W oder B, und die Menge bezeichnet die Menge Langworte, Worte oder Bytes, die zu reservieren sind. Damit zu den Beispielen, bei denen man besonders auch das Weitersetzen des Zuordnungszählers beobachten sollte. Bei den Konstanten ist ferner der abgelegte Wert zu beachten.

Wichtige Direktiven

```
  1                                  * Wichtige Direktiven
  2
  3 00000000                         .text Text- oder Programm-Segment
  4 00000000 4E42                    main trap #2 hier steht die Befehlsfolge
  5 00000002 223900000000             move.l konstant,dl
  6 00000008 23C100000000             move.l dl,work
  7 0000000E 60F0                     mainl: bra main Label mit Doppelpunkt
                                                 geschrieben, eingerueckt
  8                                  *
  9 00000000                         .Data Bereich der initialisierten Daten
 10                                  .even Ausrichtung an Wortgrenze
 11 00000000 00123456                konstant .dc.l $123456 Langwort-Konstante
 12 00000004 001500160017            konst1   .dc.w 21,22,23 Drei Wort-Konstanten
 13 0000000A 01030911                konst2   .dc.b 1,%11,@11,$11 Byte-Konstanten
 14                                  * die Argumente sind dezimal, binaer, oktal
                                       und hexadezimal
 15 0000000E 3132333435000000        konst3   .dc.l '12345' Literal im
                                     * Langwortformat, Nullen werden angehaengt
 16 00000016 7A00                    konst4   .dc 'z' Byte-Konstante 'z'
 17 00000018 7A00                    konst5   .dc "z" dito
 18 0000001A 4461732069737420        string   .dc 'Das ist ein String' String-Konstante
 18 00000022 65696E2053747269
 18 0000002A 6E67
 19                                  .even
 20 0000002C 022B                    konst6  .dc 555 Default-Einstellung auf
                                                 Wortkonstante
 21                                  *
 22 00000000                         .bss Block Storage Section fuer Variablen
 23                                  .even Ausrichtung an Wortgrenze immer
                                           empfehlenswert
 24 00000000                         work .ds.l 2 Platz fuer 2 Langworte
 25 00000008                         work1 DS.w 5 Fuenf Worte
                                           Gross-/Kleinschreibung der Direktive
 26 00000012                         work2 .ds 2 Default-Einstellung auf Wort
 27 00000016                         work3 .ds.b 15 Byte-Speicher
 28 00000026                         .even Ausrichtung an Wortgrenze
 28 00000026
 29 00000026                         work4 .ds.w 1 Ein Wort
 30                                  *
 31 00000010                         .text Weitere Programmteile koennen auch
 32                                  * unter Wiederaufnahme des
                                       Zuordnunggszaehlers
 33                                  * textlich versetzt formuliert werden
 34 00000010 4EF900000000            modul2 jmp main
 35                                  *
 36 00000016                         .end

S y m b o l   T a b l e

konst1    00000004 DATA  konst2   0000000A DATA  konst3   0000000E DATA
konst4    00000016 DATA  konst5   00000018 DATA  konst6   0000002C DATA
konstant  00000000 DATA  main     00000000 TEXT  mainl    0000000E TEXT
modul2    00000010 TEXT  string   0000001A DATA  work     00000000 BSS
work1     00000008 BSS   work2    00000012 BSS   work3    00000016 BSS
work4     00000026 BSS
```

Zu den Direktiven noch folgende Hinweise: .EVEN hat in den Segmenten DATA und BSS eine wichtige Bedeutung mit seiner Ausrichtung des Zuordnungszählers an geradzahlige Adressen. Aus der Architektur der CPU ist zu erinnern, daß Zugriffe auf Worte und Langworte zu einem Adreßfehler mit Exception führen, wenn sie auf ungerade Adressen erfolgen. Daher ist es immer empfehlenswert, .EVEN zu benutzen, vor allem, wenn Byte-Konstanten oder Speicherplatzreservierungen vom Typ Byte vorausgingen.

Wenn man bei Konstanten und Variablen keinen Zusatz zu .DC oder .DS macht, nimmt der Assembler die Operandenbreite 'Wort' an.

Der Assembler führt, wie man an den ausgedruckten Adressen sieht, eigene Zuordnungszähler für jedes Segment TEXT, DATA und BSS. Auch die ausgewiesenen Symbole werden jeweils diesen Segmenten zugeordnet. Dieses vor allem in Hinsicht auf das spätere Linken.

Ende des Quelltextes bei .END

Die Statements .END oder auch einfach END bedeuten für den Assembler: Hier ist das Ende des Quelltextes. Zeilen, die noch dahinter folgen sollten, werden ignoriert. Wenn der Assembler also im ersten Durchgang auf .END stößt, geht er sofort in den zweiten Pass über.

Kapitel 23

Sonstige Direktiven

Dieser Abschnitt gibt Hinweise auf weniger häufig benutzte Direktiven des Assemblers.

ORG Ausdruck setzt den Programmzähler auf den Wert von Ausdruck. Mit dieser Direktive erzeugt der Assembler Code für absolute Adressen. Weil unter ORG programmierte Befehle damit nicht mehr verschieblich geladen und ausgeführt werden können, wird man ORG nur ausnahmsweise benutzen. Dazu einige Beispielszeilen in der nachfolgenden Liste.

SECTION ist in den Assemblern unterschiedlich als Direktive implementiert. Die Schreibweise ist entweder SECTION mit Nummer oder mit Label. Beim erstmaligen Gebrauch richtet SECTION auf eine geradzahlige Adresse aus. Beim wiederholendem Gebrauch holt SECTION den Zuordnungszähler auf den Stand zurück, den er beim Verlassen einer benamten oder numerierten SECTION hatte. Auch dafür ein Beispiel in der Liste. Diese Direktive dient vor allem dem Linker beim Zusammenfügen von Programmteilen, die zwar an verschiedener Stelle geschrieben sind, aber mit SECTION als zusammengehörig erklärt wurden.

INCLUDE Dateiname gehört zu den wichtigeren Anweisungen mit der Bedeutung: Ein File ist einzuschließen. Wenn der Assembler diese Anweisung findet, liest er weiteren Text aus dem File 'Dateiname', und zwar bis zu dessen Ende. Erst danach fährt er ggfs. mit der Folgezeile des ersteren Quelltextes fort. In der Wir-

kung heißt das: Das mit INCLUDE gerufene File wird so verwertet, als ob sein Text im Haupttext enthalten wäre. Damit sind hervorragende Möglichkeiten gegeben, in Modulen zu programmieren und sie je nach Bedarf von einem Kommandofile aus aufzurufen. Die nachstehende Liste veranschaulicht auch das: In Zeile 2 wird das File ORG.ASM gerufen. Die Zeilen 4 ... 17 stehen als Quelltext in diesem aufgerufenen File.

Benutzung von include

```
 1                          * benutzung von include
 2                           include org.asm steht im aufrufer
 3                          * gst assembler
 4    00005555               org $5555 org setzt den zuordnungszaehler auf
                                     absolute adressen
 5    00005556 4e71         main nop
 6    00005558 4e71          nop
 7    0000555a 4ef85556      jmp main absoluter sprung
 8                          *
 9 01 00000000               section one erstes segment
10 01 00000000 03            dc.b 3 in diesem segment zunaechst 1 byte
11                          *
12 02 00000000               section two zweites segment
13 02 00000000 022b          dc.w 555
14 02 00000002 0000022c      dc.l 556
15                          *
16 01 00000001               section one wieder erstes segment
17 01 00000001 5a            dc.b 'z' wiederaufnahme der zaehlung nach
                                     section one
18                           end
```

COMMON Label Ausdruck ermöglicht getrennt assemblierten Modulen, einen Datenbereich gemeinsam zu benutzen. 'Ausdruck' bezeichnet dabei die Größe des Datenbereiches. Der Linker sorgt dafür, daß dieser Bereich die maximale Größe annimmt, die in einem der Module erklärt worden ist. Common-Bereiche werden insbesondere in der Sprache FORTRAN benutzt.

Externe Symbole, GLOBL, XDEF, XREF

Der Linker erlaubt, unabhängig voneinander assemblierte Module zu einem Programm zu verschmelzen. Wenn nun ein Modul auf Einsprungstellen und Daten eines anderen Bezug nimmt, so

muß es eine Korrespondenzmöglichkeit zwischen ihnen geben. Dem dienen die vorgenannten Anweisungen in der typischen Schreibweise **GLOBL Label1,Label12,...** In der Schreibweise dürfen also mehrere durch Kommata getrennte Symbolnamen innerhalb einer Zeile übergeben werden. Was ist nun die Wirkung? Normalerweise sind Bezeichner, die innerhalb eines Modules benutzt werden, "lokaler" Natur. Ihr Name und ihr Wert sind von außen nicht erreichbar oder verwertbar. Das erspart einerseits dem Programmierer ständig neue Wortschöpfungen, da er gleich buchstabierte Bezeichner in verschiedenen Modulen benutzen kann, ohne daß es zu einer Kollision kommt. Man kann also Bibliotheken von Modulen schreiben, ohne vor der Assemblierung gleichlautende Bezeichner beachten zu müssen.

Externe Referenzen, DRI-Assembler

```
C P / M   6 8 0 0 0   A s s e m b l e r
   1                          * Externe Referenzen, DRI-Assembler
   2 00000000                  SECTION 1
   3                          *
   4                           XREF LABEL1 Suche diese Symbole in anderen
   5                           XDEF LABEL2 Modulen
   6                           GLOBL MAIN  Das hier erklaerte MAIN wird
                                           anderen Modulen zugaenglich
   7                          *
   8 00000000 4EB900000000    MAIN JSR LABEL1 LABEL1 und LABEL2 sind
                                              externe Symbole
   9 00000006 4EB900000000     JSR LABEL2

S y m b o l   T a b l e

LABEL1    ******** EXT   LABEL2    ******** EXT   MAIN       00000000 TEXT
```

Anweisungen mit GLOBL, XDEF, XREF machen die Bezeichner eines Modules über den Linker auch für andere Module bekannt. Andererseits verweisen sie den Linker auch darauf, Bezeichner in anderen Modulen zu suchen, wenn sie nicht im aktuellen Modul erklärt worden sind (per EQU oder als Label). Je nach Assembler gibt es dabei offensichtlich Unterschiede. XREF bedeutet, daß Symbole in einem anderen Modul zu suchen sind; XDEF, daß sie anderen Modulen bekanntzugeben sind.

Externe Referenzen

```
GST 68000 Macro Assembler
   1                          * EXTERNE REFERENZEN
   2 01 00000000               SECTION EINS
   3                          *
   4                           XREF LABEL1,LABEL2 Suche diese Symbole in
                                                  anderen Modulen
   5                           XDEF MAIN          Das hier erklaerte MAIN wird anderen
                                                  Modulen zugaenglich
   6                          *
   7 01 00000000 4EBAFFFE     MAIN JSR LABEL1     LABEL1 und LABEL2 sind externe
   8 01 00000004 4EBAFFFA      JSR LABEL2         Symbole

Sections and common blocks

EINS      SECT 01 00000008

Cross-reference listing

LABEL1    XREF                4*    7
LABEL2    XREF                4*    8
MAIN      XDEF 01 00000000    5     7*
```

Die Bekanntgabe externer Symbole ist auch in Zusammenhang mit dem Aufruf des Assemblers wichtig. In der Kommandozeile dürfen einige Optionen mitgeteilt werden. Insbesondere **–U** bedeutet dabei, daß dem aktuellen Modul unbekannte ("U" wie unbekannt) Symbole als externe Symbole anzusehen sind. Diese Option –U kann dann zu folgender Situation führen: Der Assembler verläßt sich darauf, daß unbekannte Symbole in einem anderen Modul erklärt sind, und gibt daher keinen Fehlerhinweis. Schließlich wird eine irgendwie vergessene Symbolerklärung erst beim Linken aufgedeckt, zu einem Zeitpunkt also, zu dem der Programmierer dann schon vergebliche Arbeit reut. Es ist also nicht verkehrt, zunächst ohne das –U zu assemblieren, um Hinweise auf notwendige Erklärungen zu erhalten.

Gestaltung des Listbildes

Einige Direktiven betreffen das Listbild, so **TTL String** (Überschriftzeile je Seite), **SPC Ausdruck** (Leerzeilen in der Menge Ausdruck), **LIST** und **NOLIST** (Ein- und Ausschalten der Listung), **PAGE** und **NOPAGE** (Vorschub auf neues Blatt).

Bedingtes Assemblieren

Die Tätigkeit des Assemblers kann von Parametern (Ausdrucksbewertungen) abhängig gemacht werden, die ihm im Zeitpunkt des Durchlaufes zur Verfügung gestellt werden. Er soll etwas assemblieren, wenn eine Bedingung zutrifft. Andernfalls soll er einen Anweisungsblock übergehen, der sich bis zu einer Anweisung ENDC erstreckt. Solche Blöcke beginnen mit IFEQ Ausdruck, dito mit IFNE, IFLE, IFLT, IFGE, IFGT. Mnemonisch bedeuten dabei EQ = Null, NE = ungleich Null, LE = kleiner/gleich Null, LT = unter Null, GE = größer/gleich Null und GT = größer als Null. – Bedingungen dieser Art können dem Assembler besonders in einer Aufrufdatei mitgeteilt werden, damit für verschiedene Umgebungen assembliert werden kann, ohne daß man im Quelltext umfangreiche Veränderungen vornehmen muß. Die nachfolgende kleine Liste soll für "ATARI" Code erzeugen. Im Block ab IFNE steht die Alternative, falls "ATARI" nicht zutrifft.

Bedingtes Assemblieren

```
 1                          * Bedingtes Assemblieren
 2                          *
 3                          atari=0            * Bedingung mit 0 auf
                                                 'wahr' setzen
 4                          *
 5 00000000                 .text
 6                          ifeq atari         * Abfrage, ob atari
                                                 = Bedingung
 7 00000000 4E43            trap #3            * Dann Trap #3 codieren
 8                          .endc              * Ende Bedingungsblock
 9                          *
10                          ifne atari         * Alternative Bedingung abfragen
11                          trap #4            * Dann dieses trap codieren
12                          .endc              * Ende Bedingungsblock
```

Je nach Assembler wird man auch die Fähigkeit vorfinden, Programmsequenzen als ein Makro zu definieren. Wir gehen auf diese Direktiven nicht ein, weil sie sehr speziell ausgearbeitet werden müßten.

ANGEWANDTE ASSEMBLER-PROGRAMMIERUNG

ANGEWANDTE ASSEMBLER-PROGRAMMIERUNG

Kapitel 24

Das Betriebssystem

Ein kurzer Überblick macht uns mit den Begriffen vertraut, die im Zusammenhang mit dem Betriebssystem des Atari 520ST häufig genannt werden: TOS, GEMDOS, GEM, BIOS usw.

Jeder Computer braucht ein Betriebsprogramm, das ihm mindestens die Grundleistungen der Ein- und Ausgabe (E/A) zur Verfügung stellt. Die Implementierung dieser Leistungen ist von der Hardware des Computers abhängig. Es ist leicht einzusehen, daß ein Systemprogrammierer z. B. das Lesen eines Zeichens von der Tastatur verschieden anfassen muß, wenn er in einem Falle auf die Leistungen eines Tastatur-Decoderbausteines zurückgreifen kann oder wenn er, wie früher meist üblich, im anderen Falle die Leitungsmatrix einer "dummen" Tastatur abfragen muß, um festzustellen, an welchem Schnittpunkt eine Taste betätigt wurde. Die Koordinaten des Schnittpunktes muß er mit einer Tabelle in eine ASCII-Codierung umsetzen und dabei auch berücksichtigen, ob zusätzlich z. B. die Shift- oder die Control-Taste gedrückt wurde.

Das grundlegende Ein- und Ausgabesystem eines Computers nennt man **BIOS** (engl.: Basic Input Output System). Es macht die Tastatur, den Bildschirm und die Floppy Disk hantierbar. Seine 12 Funktionen sind im Abschnitt "Die BIOS-Funktionen" dargestellt. Einen etwa vergleichbaren Satz von Grundfunktionen findet man auf Computern mit CPU 68xxx und dem Betriebssy-

stem CP/M-68K von Digital Research, das auch für das BIOS des Atari Pate gestanden hat.

In der Hardware des Atari finden wir neben Bildschirm und Tastatur jedoch weitere Funktionsbausteine, die ebenfalls in ihren Grundleistungen programmiert und bedient werden wollen, so die Maus, den Printerport, die Verwaltung von Bildschirminhalten und Interrupten oder den Soundchip. Für diese Hardware wurde ein erweitertes BIOS geschrieben, das man **XBIOS** nennt.

Die nächste Schicht in der Hierarchie des Betriebssystems ist das **GEMDOS**. Es ist bereits ein von der Hardware unabhängiges logisches System, das die einfachen Funktionen des BIOS zu komplexeren zusammensetzt, z. B. Lesen eines Zeichens mit Filterung, Lesen und Schreiben einer Zeile, Floppy- und Speicherverwaltung. Bei der in diesem Buch dargestellten Assembler-Programmierung befassen wir uns vor allem mit dem GEMDOS. Hierzu noch kurze Bemerkungen: Das GEMDOS ist mit dem BDOS des Betriebssystems CP/M-68K und mit dem MS-DOS des IBM-PC vergleichbar.

GEM ist schließlich das graphische System des Atari-Computers, das z. B. das Desktop-Menue erzeugt, das Linien und Ikonen zeichnen sowie Flächen füllen kann usw. Es dient auch der interaktiven Mausbedienung. Zu den einzelnen Funktionen des GEM gehören z. T. recht komplexe Parameterübergaben, die man bequemer als in Assembler in der Sprache 'C' bewirkt. Hierfür stehen dem Systemprogrammierer entsprechende Quelltexte im optionalen Entwicklungspaket zum Atari zur Verfügung.

Der Begriff **TOS**, was immer er bedeuten soll, wird für das übergeordnete Betriebssystem benutzt, das die Leistungen der Teile BIOS, XBIOS, und GEMDOS zusammenfaßt. Dieses TOS wird beim Einschalten des Computers von Diskette geladen. Für den entsprechenden Ladevorgang ist der Computer mit einem Urlader-Programm versehen, das man auch Boot-ROM nennt. Es ist geplant, dieses TOS künftig in Festwertspeichern (ROMs) zu liefern, so daß man mit dem Einschalten des Computers so-

fort eine hantierbare Maschine vorfindet, die dann auch mehr freien (ca. 220 KB) Arbeitsspeicher hat. – Davon abgesehen wird der Anwender wohl kaum einen Unterschied bemerken, weil der Computer nach dem Einschalten heute schon prüft, ob ein entsprechendes ROM vorhanden ist. Wenn ja, dann wird zu seinem TOS-Programm verzweigt. Wenn nein, dann wird das TOS zunächst von der Diskette ins RAM geladen und dann hereingerufen.

Kapitel 25

Ein erstes Programm

Ein erstes kleines Programm zeigt uns, wie wir in Assembler auf den Atari zugehen müssen, um Leistungen seines Betriebssystems TOS zu benutzen. Wir programmieren eine Tastaturabfrage mit Echo-Ausgabe der empfangenen Zeichen. Es wird auch erklärt, wie die Assembler-Liste zu interpretieren ist.

Die Assembler-Liste

Der Assembler erzeugt aus einer Quelltextdatei eine Ausgabedatei mit speicherverschieblichem Code für den Linker und auf Wunsch auch eine Assembler-Liste. Wir betrachten jetzt diese an folgendem Beispiel.

Zeichen holen und ausgeben, GST-Assembler

```
  1                                  ** Zeichen holen und ausgeben, GST-Assembler
  2
  3                                   XDEF main
  4 01 00000000                       section one
  5 01 00000000 4EBA0008             main: jsr conin          zeichen holen
  6 01 00000004 6112                  bsr.s conout            zeichen schreiben
  7 01 00000006 4EFAFFF8              jmp main                endlose schleife
  8                                  * bios-routinen werden mit trap #13 aufgerufen
  9                                  * zeichen von konsole holen
 10 01 0000000A 3F3C0002             conin: move.w #2,-(sp)   device 2 = tastatur
 11 01 0000000E 3F3C0002              move.w #2,-(sp)         funktion 2 = conin
 12 01 00000012 4E4D                  trap #13                bios-aufruf
 13 01 00000014 588F                  addq.l #4,sp            stackpointer berichtigen
 14 01 00000016 4E75                  rts
 15                                 * zeichenausgabe auf konsole
 16 01 00000018 3F00                conout: move.w d0,-(sp)  auszugebendes zeichen
 17 01 0000001A 3F3C0002              move.w #2,-(sp)         ausgabe konsole
 18 01 0000001E 3F3C0003              move.w #3,-(sp)         funktionsnummer
 19 01 00000022 4E4D                  trap #13                bios-aufruf
 20 01 00000024 5C8F                  addq.l #6,sp            stackpointer berichtigen
 21 01 00000026 4E75                  rts
 22                                   end

Sections and common blocks

ONE       SECT 01 00000028

Cross-reference listing

CONIN          01 0000000A    5    10*
CONOUT         01 00000018    6    16*
MAIN      XDEF 01 00000000    3     5*    7
```

Im linken Drittel der Liste sehen wir 4 Spalten mit dezimalen und auch hexadezimalen Zahlen, die vom Assembler aus dem Quelltext erzeugt wurden. Der Text rechts davon ist der Quelltext. Er ist für die Listung etwas formatiert worden, spiegelt aber recht getreu das wider, was der Programmierer in das Textprogramm eingetippt hat. Wenn Sie als Leser also Programmbeispiele nachtippen und ausprobieren wollen, so halten Sie sich an den Text im rechten Teil der Listen.

Beim Quelltext machen wir gleich auf eine kleine Falle aufmerksam: Sie sehen, daß die meisten Worte um eine Spalte gegenüber derjenigen eingerückt sind, in der ein Stern steht, oder gegenüber den Worten MAIN, CONIN und CONOUT (letztere sind Label). Es hängt vom benutzten Assembler ab, ob man Befehle und Direktiven um mindestens eine Schreibspalte einrükken muß oder nicht. Der englische **GST-Assembler** verlangt es,

der **DRI-Assembler** ist da tolerant. Label dürfen immer in Spalte 1 beginnen, an ihren Namen muß man jedoch einen Doppelpunkt anhängen, wenn sie nicht in Spalte 1 beginnen. Vom Standpunkt der Systemprogrammierung ist diese kleine Komplikation vermeidbar, man muß aber ggfs. mit ihr leben.

Die erste Spalte links enthält eine Zeilennummer. Sie wird vom Assembler erzeugt, ist also nicht Teil des Quelltextes (wie bei BASIC). Die zweite Spalte ist die SECTION des GST-Assemblers. SECTION ONE wurde in Zeile 4 erklärt (der DRI-Assembler druckt keine Sektionsnummer aus). In 8 Schreibspalten rechts davon sehen wir hexadezimale Adressen, die den Zuordnungszähler des Assemblers widerspiegeln. Er beginnt mit Null als relativer Basis und wird von Befehl zu Befehl mit der Befehlslänge hochgezählt. Die vierte Spalte enthält den hexadezimalen Maschinencode der erzeugten Befehle.

Zeilen mit Stern am Beginn sind reine Kommentarzeilen. Benutzte Direktiven sind XDEF, SECTION und END (Zeilen 3, 4 und 22). Label sind MAIN, CONIN und CONOUT. Mnemonische Befehle sind JSR und BSR (zwei Formen des Unterprogrammaufrufes), JMP (Sprung), MOVE (Transport), TRAP (numerierte Programmausnahme), ADDQ (Addition, hier eines Direktwertes) und RTS (Rückkehr aus einem Unterprogramm zum Rufer in den Zeilen 5 und 6).

Das kleine Programm hat 3 logische Abschnitte: Das Hauptprogramm besteht aus den Zeilen 5 ... 7. Dort werden die Unterprogramme CONIN und CONOUT gerufen. Es schließt ein Sprung (JMP) an den Programmbeginn bei MAIN an. Das Programm läuft also in einer endlosen Schleife, aus der es nur mit Reset herauszuholen ist. Gleichwohl erlaubt es Studien während der Ausführung.

Die Abschnitte zwei und drei sind die Unterprogramme CONIN und CONOUT, die jeweils durch RTS abgeschlossen sind. Ohne die nachstehenden Erklärungen würden wir gar nicht verstehen, was sie eigentlich tun. Ihre entscheidenden Befehle sind TRAP

#13, die Aufrufe des Betriebssystems BIOS bewirken (Basic Input Output System). Für solche Aufrufe gibt es dokumentierte Regeln, wie der Aufruf vorzubereiten und was hinterher noch zu regeln ist. Das zeigen wir im Abschnitt BIOS-Aufrufe. Hier zunächst zum Prinzip:

In Zeile 10 sehen wir, daß der Direktwert 2 auf den Stack gebracht wird. Die Adressierungsart ,–(sp), bedeutet, daß der Stackpointer vor der Operation um 2 herunterzuziehen ist, weil wir 1 Wort = 2 Byte auf den Stack bringen (MOVE.W). Bei Eingaben wird hier die Gerätenummer erwartet. Die '2' steht für Systemeingabe = Tastatur. In Zeile 11 wird eine zweite '2' auf den Stack gebracht. Sie ist die Funktionsnummer für zeichenweise Eingaben. Allen BIOS-Befehlen ist eine solche Nummer zugeordnet. Der Befehl TRAP #13 führt eine Exception herbei, in der das Betriebssystem entsprechend den auf dem Stack übergebenen Parametern ausführt, was gewünscht ist.

Mit den Befehlen in den Zeilen 10 und 11 hat der Benutzer den Stackpointer um 4 (2 Worte) "verbogen". Es steht jetzt in seiner Sorgfalt, wieder für Ordnung zu sorgen. Das geschieht nach dem Trap in Zeile 13: Zum Inhalt des Stackpointers wird der Direktwert 4 addiert, womit er auf seinen früheren Wert zurückkehrt. – Nun wird man fragen, wo denn das empfangene Zeichen geblieben ist. Bei diesem Aufruf steckt es gemäß den BIOS-Regeln im Datenregister D0. Es wird nach den Regeln für die Funktion 3 in Zeile 16 auf den Datenstack gebracht, wenn jetzt die Ausgabe vorbereitet wird. Hinterhergeschickt wird eine '2' als Bezeichnung der Systemausgabe und eine '3' für die Funktionsnummer der zeichenweisen Ausgabe (Zeilen 17 und 18). Trap #13 vollzieht jetzt die Ausgabe per BIOS. Wir sehen die gedrückte Taste als Zeichen auf dem Bildschirm. – Nun haben wir in den Zeilen 16 ... 18 drei Worte (6 Bytes) auf den Stack gebracht. In Zeile 20 müssen wir den Stackpointer daher um 6 berichtigen.

Es sind die numerierten Traps, die uns per Assembler-Programm ein Eindringen in Leistungen der Maschine möglich ma-

chen. Weitere Abschnitte des Buches gehen daher auf solche Funktionsaufrufe ein und auf das, was man bei ihnen beachten muß. Hier noch kurz einige Hinweise: Auf dem Atari sind folgende vier (von 16) numerierten Traps benutzbar:

TRAP #1 GEMDOS-Aufrufe höherer Dienstleistungen
TRAP #2 GEMDOS-Erweiterungen
TRAP #13 BIOS-Aufrufe einfacher Dienstleistungen
TRAP #14 XBIOS, erweiterte einfache Dienstleistungen

Warum numerierte Traps?

Betriebssysteme sind heute bereits so große gedankliche Gebilde geworden, daß kaum noch eine Einzelperson einen vollständigen Überblick behalten kann, welche Assemblerroutine was erledigt und wie die Ein- und Ausgabebedingungen dazu sind. Für alle Betriebssysteme, Sprachen und Compiler werden im Verlaufe neuere Versionen herausgebracht. Wenn der Anwender sich dabei immer auf neue absolute Adressen eines Programmes einstellen müßte, dann bräche schnell das Chaos aus. Auch Systemprogrammierer sind auf eine gleichbleibende Art der Prozeduraufrufe angewiesen, die von der vorliegenden Version oder von Adressen unabhängig ist. Die notwendige Normierung erfolgte für Systeme mit CPU 68xxx seitens Digital Research über die Traps. Vorbilder gab es in den Betriebssystemen CP/M und MS-DOS. Traps führen zu einer Programmunterbrechung, zu einer Exception, für die per Betriebssystem die Adresse einer Routine vorgehalten wird, die die Ausnahme bearbeitet. Bei den numerierten Traps wird dabei auf dem Stack nachgesehen, welche Parameter vom Anwender dorthin gebracht wurden, insbesondere, welche Funktionsnummer er aufgerufen hat. Von ihr werden die weiteren Handlungen abhängig gemacht.

Die Traps sind damit eine Art Schlüsselsystem, das dem Anwender zur Maschine geliefert wird. Er braucht sich nicht um die

Kerben auf den Schlüsseln zu kümmern oder darum, ob er sein Betriebssystem von der Floppy laden mußte oder ob es in Festwertspeichern verankert ist. Der Schlüssel und seine gleichbleibende Hantierungsvorschrift sorgen dafür, daß man Leistungen vom System erhält. Der zeitliche Aufwand zur Abwicklung solcher Prozeduren ist größer, als wenn man einfach programmieren sollte: JSR Routine. Gedanklich ist das System jedoch einfacher. – Nun könnte man auf den Gedanken kommen, daß man mit einem zuverlässigen Disassembler-Listing in der Hand die Dinge um Mikrosekunden schneller machen könnte. Das geht natürlich. Gleichwohl wird davor gewarnt, weil solche Tricks den Anwender mit dem Rest der Welt und mit der nächsten Version des Betriebssystems inkompatibel machen würden. – Assemblerprogrammierung sollte nicht dazu verleiten, Probleme unkonventionell zu erledigen.

Wie assembliert man nun?

Der Leser sollte den Versuch machen, den vorstehenden Programmvorschlag abzutippen und auf dem Atari zum Laufen zu bringen. Die notwendigen Handgriffe möchten wir nicht doppelt darstellen und verweisen ihn daher auf den Abschnitt "Vom Text zum Programm". Gleichwohl noch ein Tip: Wenn Sie den Text mit dem **GST-Assembler und -Linker** bearbeiten, dann erhalten Sie ein sofort ausführbares Programm und keine Fehlermeldung. Wenn Sie das entsprechende **DRI-Paket** benutzen, dann bemängelt der Linker, daß ihm das Symbol _MAIN fehlt. Anwenderprogramme sollen nach der Vorkehrung im Entwicklungspaket bei diesem Label beginnen. Ändern Sie daher in den Zeilen 3, 5 und 7 das Label MAIN zu _MAIN (mit Unterstreichungsstrich) ab. Und in Zeile 4 schreiben Sie bitte SECTION 1. Es gibt also kleine Unterschiede zwischen den Softwarepaketen. Erwähnenswert mag noch sein, daß der GST-Assembler wiederum keine Label akzeptiert, die mit einem Unterstreichungsstrich beginnen.

Die Programmerprobung

Bei den ersten Versuchen mit dem Zusammenspiel von Textprogramm, Assembler und Linker treten sicher einige Fehlermeldungen auf, die Sie beachten sollten. Wenn Sie die Ursachen untersuchen, dann achten Sie bitte zunächst auf Tippfehler und vielleicht vergessene Befehle und Direktiven. In einer neuen Umgebung des Programmierens treten natürlich auch Hantierungsfehler ein. – Wenn Sie nun zu einem fehlerfreien ausführbaren Programm von Typ **Name.PRG** gekommen sind, dann klicken Sie sein Bild mit der Maus an und melden die Anwendung im Auswahlkasten "Optionen" an. Es erscheint ein Dialogfeld wie auf Seite 48 des Atari-Handbuches dargestellt. Tippen Sie auf "TOS" und dann auf "OK". Es erscheint wieder das Desktop-Menue. Klicken Sie jetzt bitte mit der Maus zweimal auf das Bild des Programmes, um es zu starten. Es wird also gestartet, wie andere Programme auch. Nach dem Programmstart verschwindet das Bild des Desktop, der Bildschirm wird weiß und in der linken oberen Ecke erscheint ein blinkender Cursor. Wenn Sie jetzt die Tastatur betätigen, dann erscheinen die getippten Zeichen auf dem Bildschirm. Sie haben es also geschafft, ein erstes kleines Programm zum Laufen zu bringen! Bei weiteren Assemblierungen werden Sie dann schnell an Übung gewinnen.

Nach kurzer Erprobung mag es langweilig sein, nur das Echo getippter Zeichen auf dem Bildschirm zu finden. Machen Sie doch nun bitte Experimente und ein Protokoll dazu, wie sich die Steuertaste ESC (Escape) als Vortaste zu Buchstaben und wie sich das gleichzeitige Betätigen der Taste Control und anderer Tasten auswirkt, wobei es auch einen Einfluß hat, wenn zugleich auch die Taste Shift gedrückt wird. Sie werden schnell feststellen, daß die Zeichenausgabe unserer Routine CONOUT bei Benutzung der Steuertasten eine den Bildschirm editierende Wirkung hat, die man in Anwendungsprogrammen verwenden kann. Hier die Zusammenfassung:

Mit Escape als Vortaste:

A Cursor eine Zeile nach oben
B Cursor eine Zeile tiefer
C Cursor rechts
D Cursor links
E Bildschirm löschen und Cursor Home
H Cursor Home
I Cursor in gleicher Spalte aufwärts, Text nach unten scrollen, wenn man in der obersten Zeile ist
J Löschen des Schirmes ab Cursor-Position
K Löschen bis Zeilenende
L Zeile einfügen (Insert)
M Zeile löschen
Y Cursor auf Zeile und Spalte setzen, und zwar entsprechend dem ASCII-Wert der beiden Folgezeichen. Der Space hat dabei den Wert 0, das Ausrufungszeichen den Wert 1 usw.
b Vordergrundfarbe gemäß Folgezeichen setzen, wobei dessen niederwertige 4 Bits verwertet werden
c Hintergrundfarbe setzen, Folgezeichen wie bei b
d Vom Bildbeginn bis zum Cursor löschen
e Cursor sichtbar machen
f Cursor unsichtbar machen
j Cursorposition merken
k Cursor an gemerkte Stelle zurücksetzen
l Zeile löschen, Cursor an ihren Anfang setzen
o Vom Zeilenbeginn bis Cursor löschen
p Inverse Zeichendarstellung einschalten
q Inverse Zeichendarstellung ausschalten
v Automatischer Übergang vom Zeilenende auf die Folgezeile
w Anhalten der Zeichenausgabe in der letzten Spalte

Zusammen mit der Taste Control, ohne oder mit Shift:

h	Backspace
i	Tabulator, 8 Spalten
j	Linefeed, Vorschub in neue Zeile
k	Cursor in Folgezeile setzen
m	Carriage Return, Rücklauf an den Zeilenanfang

Kapitel 26

Die BIOS-Funktionen

Das BIOS ist das grundlegende Ein- und Ausgabesystem des Computers. Seine Funktionen sind numeriert und werden mit dem Befehl TRAP #13 angefordert, nachdem man zuvor Parameter und die Funktionsnummer auf den Stack gebracht hat. Ergebnisse erhält man im Datenregister D0. Nach einem Aufruf muß man die Veränderung des Stackpointers rückgängig machen. – Als Assembler-Programmierer wird man kaum alle Funktionen benutzen.

BIOS ist die Abkürzung für **B**asic **I**nput **O**utput **S**ystem. Es besorgt die einfachen Ein- und Ausgaben für Tastatur, Bildschirm, Diskette und serielle Schnittstelle. Es gibt Ähnlichkeiten zum BIOS des Betriebssystems CP/M-68K, die Parameterübergabe erfolgt jedoch auf dem Stack. BIOS-Aufrufe sollten nicht tiefer als drei ineinander geschachtelt werden. Höhere Dienstleistungen erhält man mit den GEMDOS-Funktionen. Der prinzipielle Gebrauch der BIOS-Funktionen wird im Abschnitt "Ein erstes Programm" dargestellt.

Die Nummern und Namen der BIOS-Funktionen sowie ihre Parameter und Besonderheiten werden nachfolgend besprochen. Der Anhang enthält eine tabellarische Übersicht zu den Funktionen. Weiterhin: Im Abschnitt "Eine BIOS-Toolbox" findet man für alle wichtigen Leistungen des BIOS einen Assembler-Quelltext.

0 GETMBP Zeiger.L

(anschließende Stack-Korrektur 6)

Die Funktion füllt einen Memory Parameter Block an der Adresse 'Zeiger' mit drei Systemvariablen (freier Speicher, besetzter Speicher, Wanderzeiger). Jede dieser Systemvariablen ist wiederum Zeiger auf eine Struktur aus vier Langworten: Link zur nächsten Struktur (oder 0), Startadresse und Menge freier Bytes sowie Prozeßkennzeichner (oder 0). Einen solchen Block für das TOS finden wir ab Adresse $48E, er wird bei der Initialisierung des Systems mit GETMBP angelegt.

1 BCONSTAT Gerätenummer.W

(Stack-Korrektur 4)

Eingabestatus des Eingabegerätes mit Nummer holen. Das Register D0.L ist 0, wenn kein Zeichen vorliegt bzw. $FFFFF, wenn mindestens ein Zeichen anhängt. Es gelten folgende Geräte-Nummern:

0 Drucker am parallelen Port (Centronics)
1 AUX, Schnittstelle RS232 (Auxiliary Device)
2 CON, Konsole, Tastatur und Bildschirm
3 MIDI-Schnittstelle
4 Tastatur-Port (Atari-Erweiterung)

BCONSTAT kann nicht auf die Geräte 0 und 4 angewandt werden. Im Abschnitt "Eine BIOS-Toolbox" ist für jedes Gerät gleich ein fertiges Modul formuliert worden, und zwar für die BIOS-Funktionen BCONSTAT, BCONIN, BCONOUT und BCOSTAT.

2 BCONIN Gerätenummer.W

(Stack-Korrektur 4)

Ein Zeichen vom Gerät mit Nummer (wie bei BCONSTAT) einlesen. Dabei warten, bis ein Zeichen vorliegt. Ergebnis im Register

D0.L ablegen. Das höherwertige Wort im Register ist 0, außer bei der Tastatureingabe (Gerät 2). Hier ist in den Bitstellen 16 ... 23 der IBM-kompatible Tastaturcode, im niederwertigen Byte (Bits 0 ... 7) das ASCII-Zeichen. Und es gibt noch eine Spielart in Abhängigkeit von Bit 3 in der Systemvariablen CONTERM (Adresse $482): Ist es 1, dann spiegeln die Bits 24 ... 31 von D0.L den Shift-Modus wider (s. a. Funktion KBSHIFT). Und noch eine Besonderheit zur Variablen CONTERM: Bit 0 schaltet die Glocke ein/aus, Bit 1 den Tastenrepeat und Bit 2 das Klicken der Tasten. – BCONIN kann nicht auf das Gerät 4 angewandt werden.

3 BCONOUT Gerätenummer.W, Zeichen.W

(Stack-Korrektur 6)

Ein Zeichen auf das Gerät mit Nummer (wie bei BCONSTAT) ausgeben. Warten, bis die Ausgabe beendet ist. Beispiel im Abschnitt "Ein erstes Programm".

4 RWABS Rwflag.W, Buffer.L, Menge.W, Startsektor.W, Device.W

(Stack-Korrektur 14)

Lesen oder Schreiben logischer Sektoren ab logischem Startsektor in der Anzahl 'Menge' von/auf Device mit Nummer. Die Datenbewegung findet ab der Langwortadresse in Buffer statt. Es bedeuten im Rwflag: 0 = Lesen, 1 = Schreiben, 3 = Lesen ohne Beachtung eines Diskettenwechsels und 4 = Schreiben ohne Beachtung eines Diskettenwechsels.

Der Parameter 'Device' hat folgende Bedeutung: 0 = Laufwerk A, 1 = Laufwerk B, 2 und höher = Harddisk. – Man beachte hier, wie auch bei anderen Funktionen, daß der letztgenannte Parameter rechts (hier die Devicenummer) zuerst auf den Stack gebracht werden muß, dann der davorstehende usw. Als letztes

wird die Funktionsnummer auf den Stack gebracht, dann das Trap aufgerufen.

5 SETEXC Vektornummer.W, Vektor.L

(Stackberichtigung 8)

Setzen eines Exception-Vektors bei logischer Vektornummer. Im Abschnitt Exceptions wurden die Vektoren für Programmausnahmen behandelt. Die Vektoren des 68000 sind mit 2 ... 255 numeriert. Für das GEMDOS sind die drei darauf folgenden Vektoren belegt. SETEXC transportiert ein Adreß-Langwort in die Vektortafel im RAM. Die Speicheradresse ist dabei 4*logische Adresse, weil Vektoren immer Langworte sind. – Diese Funktion liefert in D0.L den Vektor ab, der zuvor an der betreffenden Stelle verankert war.

6 TICKCAL

(Stackberichtigung 2)

Liefert die Kalibrierung des System-Timers in Millisekunden ab.

7 GETBPB Devicenummer.W

(Stackberichtigung 4)

Diese Funktion liefert einen Zeiger als Langwort ab, der auf einen BIOS Parameterblock (BPB) zum spezifizierten Laufwerk zeigt (0 = Drive A, 1 = Drive B usw.). In D0.L steht jedoch 00, wenn keine Zuordnung erfolgen konnte. Ein solcher BPB beschreibt folgende Attribute des Laufwerkes jeweils in einem Wort: Bytes pro Sektor (512 bei Floppys), Clustergröße (2 bei Floppys), Bytes pro Cluster (dann 1024), Länge des Directory, Zahl der Sektoren für die File Allocation Table (FAT), Sektornummer des folgenden FAT, Nummer des ersten Datensektors und Anzahl der Datencluster je 1 KB.

8 BCOSTAT Gerätenummer.W
(Stackberichtigung 4)

Wie in Funktion 1 wird geprüft, ob ein Gerät der Zeichenausgabe bereit ist. Wenn ja, dann wird in D0 der Wert –1 abgeliefert, sonst 0.

9 MEDIACH Drivenummer.W
(Stackberichtigung 4)

Es wird festgestellt, ob zwischenzeitlich die Diskette gewechselt wurde: 0 = kein Wechsel, 1 = Wechsel möglich, 2 = Wechsel ist sicher.

10 DRVMAP
(Stackberichtigung 2)

In den 32 Bits des Registers D0.L wird als binäre Strichliste mitgeteilt, welche Drives (0 ... 31) aktiv sind. Die Systemvariable in Adresse $4C4, die das verwaltet, heißt DRVBITS. In den Bits 0 und 1 finden wir normal Drive A und B aktiviert, so daß sich das Register dann zu 3 ausliest.

11 KBSHIFT Modus.W
(Stackberichtigung 4)

Bei positivem Parameterwort Modus werden Shift-Bits gesetzt, bei negativem werden Shift-Bits gemeldet. Dabei haben die Bits in D0 folgende Bedeutung: 0 = rechte und 1 = linke Shift-Taste, 2 = CTRL-Taste, 3 = ALT-Taste, 4 = Caps Lock, 5 = rechter und 6 = linker Knopf der Maus.

Kapitel 27

Eine BIOS-Toolbox

In diesem Abschnitt beginnen wir mit der Einrichtung eines "Werkzeugkastens". Für die weiteren Arbeiten werden immer wieder benötigte Programm-Module als Assembler-Quelltext formuliert, die der Leser als Vorlage zum Aufbau einer eigenen Quelltext-Bibliothek benutzen sollte. Solche Module können danach mit der Assembler-Direktive **'INCLUDE Filename'** in weitere Quelltexte eingebunden werden. Damit spart man sich künftig Arbeiten des Eintippens und der Fehlersuche. – Diese Toolbox wird in den folgenden Abschnitten nach und nach erweitert.

Planungsfragen

Bei den Programmbeispielen in den weiteren Abschnitten dieses Buches wird immer wieder auf die grundlegenden Leistungen des BIOS und des GEMDOS zurückgegriffen. Es würde den Text unnötig aufblähen, wenn wir bereits erklärte Lösungen immer wieder dokumentieren müßten. Wir benutzen daher die Fähigkeiten des Assemblers, Files zu verbinden, und zwar mit der Direktive 'INCLUDE Filename'. Durch sie werden Quelltext-Dateien so behandelt, als wenn ihr Text im aufrufenden Programmtext stünde. Als Programmierer haben wir den Vorteil, daß wir auf archivierte und ausgetestete Module zurückgreifen können. Und fast wichtiger noch: Wir können mnemonisch z. B. formulieren 'JSR BCONSTAUX' und erhalten damit immer im Register D0 den Status der RS232-Schnittstelle. Wenn er negativ ist, dann ist die

Schnittstelle bereit, wenn er 00 ist, dann nicht. – Wenn Sie die nachfolgende Programm-Liste betrachten, dann werden Sie feststellen, daß die Grundbefehle des BIOS bereits für die möglichen Geräte des Systems aufgefächert sind, um eine bequemere Programmierung zu ermöglichen. Statt einfach BCONSTAT gibt es ein BCONSTAUX (serielle Schnittstelle), ein BCONSTCON (für den Bildschirm) und ein BCONSTMIDI (MIDI-Schnittstelle).

Es gibt natürlich auch andere Möglichkeiten, eine Bibliothek zu benutzen. So könnte man auch vom Assembler vor-übersetzte Files beim Linken einbeziehen. In diesem Falle hätte man den Vorteil der Geschwindigkeit, würde jedoch keine komplette Assembler-Liste erhalten. Auch müßte man mehr mit externen Symbolen arbeiten.

Bei der Planung einer Modul-Bibliothek gibt es weiter Gesichtspunkte: Unterscheidet der Assembler zwischen Groß- und Kleinschreibung bei den Symbolen? Darf der Unterstreichungs-Strich in Namen benutzt werden? Und: Wieviele Zeichen in Namen sieht der Assembler als signifikant an? Im allgemeinen werden es 8 Zeichen sein. Wenn man das nicht berücksichtigt, erhält man die Fehlermeldung, daß Symbole mehrfach definiert wurden. Zum Beispiel schauen wir auf die nachfolgende Liste: BCONOUTPRNT und BCONOUTAUX unterscheiden sich erst im 8. Zeichen ab P im -PRNT bzw. ab A im -AUX. Das reicht gerade noch aus, wenn man aus mnemonischen Gründen den längeren ganzen Namen benutzt.

Die Direktive INCLUDE sollte man in einem Programmtext nicht am Beginn benutzen, weil Assembler hier oft den Beginn des Hauptprogrammes erwarten. Modul-Unterprogramme gehören in diesen Fällen weiter hinten in den Quelltext. Mit den Direktiven LIST und NOLIST können wir weiterhin steuern, ob die herangezogene Modul-Bibliothek immer wieder gelistet werden soll. Dieser Hinweis führt nach einiger Übung vielleicht zu einer Abfolge wie: Text des Hauptprogrammes, NOLIST, INCLUDE Module, LIST, weiteres Programm.

Zur Programm-Liste

Nach den Erklärungen in den Abschnitten "Ein erstes Programm" und in "Die BIOS-Funktionen" dürften die gelisteten Unterprogramme leicht verständlich sein: Alle Programmteile sind vom Hauptprogramm des Anwenders mit dem Befehl 'JSR Name' als Unterprogramme aufzurufen. Sie befördern Parameter auf den Stack; der letzte ist immer die Funktionsnummer als Wort. Sie lösen dann das Trap #BIOS (Nr. 13) aus, berichtigen darauf den Stackpointer mit dem Befehl ADDQ.L und kehren danach mit RTS zurück. In einigen Fällen wird zuvor noch mit TST D0 geprüft, ob die Funktion im Register einen Wert von 00 oder einen negativen Wert abgeliefert hat. Im aufrufenden Programm kann damit an den Befehl JSR sofort ein Verzweigungsbefehl (BEQ-BNE oder BPL-BMI) angeschlossen werden.

BIOS-Routinen zur Einbindung

```
*     BIOS-Routinen zur Einbindung
*     File BIOSFUN

BIOS       equ        13

*     getmbp initialisiert einen ersten Memory Parameter Block MPB
*     Setzen der Transient Program Area, hier nicht ausgefuehrt

*     bconstat prueft den Status zeichenweise arbeitender
*     E/A-Geraete. Wir trennen hier gleich fuer verschiedene Geraete

bconstaux                                ;fuer RS232-Schnittstelle, Geraet 1
          move.w     #1,-(sp)            Geraete-Nr.
          bra.s      bconstat            branch always
bconstcon                                ;fuer den Bildschirm
          move.w     #1,-(sp)            Geraet
          bra.s      bconstat
bconstmidi                               ;fuer MIDI-Port
          move.w     #3,-(sp)

bconstat  move.w     #1,-(sp)            Funktionsnummer
          trap       #BIOS               Aufruf
          addq.l     #4,sp               Stack berichtigen
          tst.l      D0                  Status ist in D0, 00=nicht bereit,
          rts                            negativ=bereit
*
*     bconin wartet und holt ein Zeichen von der Eingabe
*     Es wird gleich wieder fuer Geraete aufgeteilt

bconinprnt                               ;fuer Printer
          move.w     #0,-(sp)            Geraet
          bra.s        bconin
```

Fortsetzung

BIOS-Routinen zur Einbindung (Fortsetzung)

```
bconinaux                                   ;fuer RS232-Schnittstelle
          move.w    #1,-(sp)
          bra.s     bconin                  Sammelpunkt
bconincon                                   ;Bildschirmgeraet
          move.w    #2,-(sp)
          jmp      bconin
bconinmidi                                  ;MIDI-Port
          move.w    #3,-(sp)
bconin    move.w    #2,-(sp)                Sammelpunkt Funktionsaufruf
          trap      #BIOS
          addq.l    #4,sp                   Stack berichtigen
          rts                               D0.L enthaelt ASCII und Scan-Code

*    bconout gibt ein Zeichen aus und wartet,
*    bis es abgenommen ist
*    Das Zeichen muss beim Aufruf bereits in Register D0 sein

bconoutprnt                                 ;fuer Printer
          move.w    #0,-(sp)                Geraet
          bra.s     bconout
bconoutaux                                  ;fuer RS232-Schnittstelle
          move.w    #1,-(sp)
          bra.s     bconout                 Sammelpunkt
bconoutcon                                  ;Bildschirmgeraet
          move.w    #2,-(sp)
          bra.s     bconout
bconoutmidi                                 ;MIDI-Port
          move.w    #3,-(sp)
          bra.s     bconout

bconoutkbd                                  ;Ausgabe an Keyboard
          move.w    #4,-(sp)
bconout   move.w    #3,-(sp)                Sammelpunkt Funktionsaufruf
          trap      #BIOS
          addq.l    #4,sp                   Stack berichtigen
          rts

*         bcostat zeigt an, ob ein Ausgabegeraet abnahmebereit ist
*         Wenn ja, dann ist D0.L negativ, wenn =00, dann warten

bcostatprnt                                 ;fuer Printer
          move.w    #0,-(sp)                Geraet
          bra.s     bcostat
bcostataux                                  ;fuer RS232-Schnittstelle
          move.w    #1,-(sp)
          bra.s     bcostat                 Sammelpunkt
bcostatcon                                  ;Bildschirmgeraet
          move.w    #2,-(sp)
          bra.s     bcostat
bcostatmidi                                 ;MIDI-Port
          move.w    #3,-(sp)
          bra.s     bcostat
bcostatkbd                                  ;Ausgabe an Keyboard
          move.w    #4,-(sp)
bcostat   move.w    #8,-(sp)                Sammelpunkt Funktionsaufruf
          trap      #BIOS
          addq.l    #4,sp                   Stack berichtigen
          tst.l     D0                      Gleich den Status holen
          rts

*         getbpb erbringt einen Zeiger in D0.L
*         auf einen BIOS-Parameter Block (BPB)
*         fuer einen Drive mit Nummer,
*         die im Register D0.W bereits beim Aufruf stehen muss
```

Fortsetzung

BIOS-Routinen zur Einbindung (Fortsetzung)

```
getbpb    move.w    D0,-(sp)          D0 mit Drive-#, A=0, B=1 usw.
          move.w    #7,-(sp)          Funktionsnummer
          trap      #BIOS             Aufruf
          addq.l    #4,sp             Stack berichtigen
          rts                         Wenn D0=00, dann Fehler

*         D0 zeigt jetzt auf einen Block von 9 Worten, in dem man findet:
*         Sektorgroesse in Bytes/Clustergroesse in Sektoren, Clustergroesse
*         in Bytes, Sektoren-# im Directory, File Allocation Table (FAT
*         in Sektoren), Sektor des folgenden FAT, Sektor des 1. Daten-
*         Clusters, # Daten-Clusters auf der Diskette und Flags
*
*    mediach = media cachanged ?
*    Fuer einen Disketten-Drive kann festgestellt werden,
*    ob die Diskette zwischenzeitlich ausgetauscht wurde
*    Beim Aufruf muss in D0.W die Drive-Nummer uebergeben worden sein
*    Dabei ist: 0= Drive A, 1=B usw.
*
mediach   move.w    D0,-(sp)          Uebergebene Geraete-Nr.
          move.w    #9,-(sp)          Nr. Funktion
          trap      #BIOS             Aufruf
          addq.l    #4,sp             Stack berichtigen
          rts                         In D0 ist abzulesen:
*                                     0=definitiv kein Wechsel
*                                     1=moeglicher Wechsel
*                                     2=sicher gewechselt

*    drvmap liefert ein Langwort in D0.L mit einem Bitvektor, der fuer
*    jedes angemeldete Laufwerk eine binaere '1' enthlt
*    Normal ist eine hex 3 fuer angeschlossene Laufwerke A und B

drvmap    move.w    #10,-(sp)         Funktionsnummer
          trap      #BIOS             Aufruf
          addq.l    #2,sp
          rts                         Jetzt Wert/Device abfragen in D0

*    Uebrige BIOS-Funktionen wie rwabs, setexec und kbshift
*    Sollten von Fall zu Fall mit Parameteruebergabe nach Bedarf
*    implementiert werden
```

Kapitel 28

Ein GEMDOS-Toolkit

Dieser Abschnitt enthält die Quelltexte nebst Erklärungen für 15 wichtige Ein- und Ausgabefunktionen des GEMDOS sowie bereits eine zusammengesetzte Funktion für die Ausgabe eines Carriage Return nebst Linefeed auf den Bildschirm. Mit diesen Funktionen können wir die Systemeinheiten Tastatur, Bildschirm, Drucker und die serielle RS232-Schnittstelle zeichen-und stringweise bedienen. Andere Buchabschnitte gehen dann auch auf Floppy-Funktionen ein.

GEMDOS enthält die höheren Funktionen des Atari-Betriebssystems, die bereits von der Hardware unabhängig sind und die auf dem für die Hardware angepaßten BIOS fußen. Es gibt Ähnlichkeiten mit dem BDOS von Digital Research, die Übergabeprozeduren sind jedoch anders. GEMDOS darf nicht mit dem graphischen Bildschirminterface GEM verwechselt werden.

Im Anhang sind neben den BIOS-Funktionen auch die GEMDOS-Funktionen vollständig aufgeführt. Wir haben dabei eine Gruppierung wie folgt vorgenommen:

- Einfache lesende Funktionen und
- Einfache schreibende Funktionen für die Systemeinheiten
- File-orientierte Funktionen
- Prozeß beenden
- Diverse Parameter setzen/holen

Das nachfolgende Modul eines Quellprogrammes enthält alle Funktionen für die ersten beiden Gruppen sowie mit TERM die Möglichkeit, einen Prozeß abzubrechen. Beispiele für die Anwendung von Funktionen aus den übrigen Gruppen finden sich bedarfsweise in anderen Abschnitten des Buches. – Dem Leser wird empfohlen, diese Quellprogramme durchaus mit ihren Kommentaren auf Diskette zu speichern, damit sie ihm bei den Programmierversuchen mit der Direktive INCLUDE oder per Linker arbeitssparend zur Verfügung stehen. Im Abschnitt "Eine BIOS-Toolbox" wurden dazu Überlegungen mitgeteilt. Im weiteren Verlauf des Buches werden die hier aufgeführten Funktionen mit ihrem mnemonischen Namen benutzt, ohne daß die Programmbeispiele immer wieder eine Liste dieser Unterprogramme enthalten würden. Sie werden dann also als bekannt und dokumentiert behandelt.

GEMDOS-Funktionen werden mit dem TRAP #1 aufgerufen. Wie bei BIOS-Aufrufen müssen zuvor Parameter auf den Stack gebracht worden sein. Zuoberst muß dort die Funktionsnummer als ein Wort stehen. Nach der Ausführung des Traps steht es in der Verantwortung des Programmierers, daß er den Stackpointer auf den Stand zurückbringt, der vor der Parameterübergabe bestanden hatte. Soweit die nachstehenden Funktionen etwas im Register D0 zurückliefern, wurde gleich der Befehl 'TST D0" programmiert, so daß man im aufrufenden Programm in Abhängigkeit von den erzielten Bedingungen logisch verzweigen kann, denn der Befehl RTS (Rückkehr vom Unterprogramm) zerstört das Bedingungscode-Register CCR nicht. – Die nachfolgenden Module sind mit 'JSR Modul' aufzurufen. Sie berücksichtigen bereits die hier genannten Übergabeprozeduren. Nun einige Bemerkungen im einzelnen:

0 TERM

Das laufende Programm wird abgebrochen, man kehrt zum aufrufenden Programm zurück, im allgemeinen also zum TOS des Atari. In diesem Falle erscheint wieder der bekannte Desktop auf

dem Bildschirm. Man beachte, daß man TERM mit einem JMP-Befehl anspringen muß, denn ein Unterprogrammaufruf würde den bisher (hoffentlich sauber verwalteten) Stackpointer verändern.

1 CONIN
(Stackberichtigung 2)

Holen eines ASCII-Zeichens von der Tastatur nach D0.B. In den Bits 8 ... 15 von D0 wird unter gewissen Umständen der Tastatur-Scancode abgeliefert, der hier jedoch nicht dokumentiert ist.

2 CONOUT Zeichen in D0
(Stackberichtigung 4)

CONOUT ist das Gegenstück zu CONIN mit der Ausgabe eines einzelnen ASCII-Zeichens an den Bildschirm.

3 AUXIN
(Stackberichtigung 2)

Holen eines Zeichens von der seriellen Schnittstelle z. B. im Terminalbetrieb oder im Rechenverbund.

4 AUXOUT und 5 PRTOUT Zeichen in D0
(Stackberichtigung 4)

Diese Funktionen wirken wie CONOUT auf die serielle Schnittstelle RS232 bzw. auf die parallele Centronics-Schnittstelle des Druckers.

6 CONIO
(Stackberichtigung 4)

Diese Funktion hat mehrere Gesichter: Wenn man sie mit dem Parameter $FFFF aufruft, dann erhält man im Register D0 eine 00 zurück, wenn keine Taste betätigt wurde. Anderenfalls erhält man das ASCII-Byte der Taste. Für diese Funktion ist aufzurufen 'JSR CONIOIN'. – Andererseits können wir ein ASCII-Byte ausgeben, wenn wir es im Register D0 als Wort mit führenden Nullen vor dem Aufruf 'JSR CONIO' übergeben.

7 DIRCON und 8 DIRCONCTRL
(Stackberichtigung 2)

Beide Funktionen lesen ein Zeichen von der Tastatur in das unterste Byte des Registers D0. DIRCON erzeugt kein Echo auf dem Bildschirm, während DIRCONCTRL Control-Codes an den Bildschirm zur Steuerung weitergibt.

9 PRINTLINE Zeiger in D0
(Stackberichtigung 6)

Diese Funktion ist das Gegenstück zu '10 READLINE'. Register D0 muß jedoch auf den Beginn des Strings selbst zeigen (nicht auf seinen Längenparameter). Daher erfolgt der Abbruch der Ausgabe auch nicht in Abhängigkeit von einem Längenparameter, sondern beim Antreffen einer hexadezimalen 00.

10 READLINE Zeiger in D0
(Stackberichtigung 6)

Dieser Funktion muß im Register D0 vor dem Aufruf ein Adreßzeiger mitgeteilt werden, wohin der von der Tastatur mit CR-Abschluß terminierte String abzulegen ist (Adresse eines Eingabebuffers). Normalerweise wird das ein Buffer sein, den man im

Programm in der Block-Storage Section (BSS) als Array definiert hat. Man muß weiterhin daran denken, daß das erste Byte des Buffers seine maximal zulässige Länge festlegt, die man also vor dem Aufruf initialisiert haben muß, z. B. mit MOVE.B #255,BUFFER. Nach der Funktionsausführung steht im 2. Byte des Buffers die erreichte tatsächliche Eingabelänge. Im 3. Byte (Buffer + 2) beginnen die Daten des eingegebenen Strings. Das letzte Byte im Buffer ist das empfangene CR.

11 CONINSTAT und 18 AUXINSTAT
(Stackberichtigung 2)

Diese Funktionen prüfen, ob in den Eingabepuffern der Tastatur oder der seriellen Schnittstelle Zeichen anhängig sind.

16 CONOUTSTAT 17 PRINTSTAT und 19 AUXOUTSTAT
(Stackberichtigung 2)

Diese Funktionen prüfen, ob die Ausgabegeräte Bildschirm, Drucker oder RS232 für die Entgegennahme eines Zeichens bereit sind.

CRLF

Dieses Unterprogramm gehört nicht zum GEMDOS, es steht bereits eine Ebene höher. Unterprogramme wie z. B. PRINTLINE bleiben bei der Ausgabe mitten in der Zeile stehen, wenn der Begrenzer erkannt wurde. In den meisten Fällen möchte man weitere Textausgaben jedoch in einer neuen Zeile finden. Dazu ruft man JSR CRLF.

GEMDOS-Funktionen zur Einbindung

```
*     GEMDOS-Funktionen zur Einbindung
*     File GEMDOSFN
*     Ergebnisse im allgemeinen im Register D0.L, mit Ausnahmen
*     Zur evtl. Fehlerpruefung D0.W auswerten

GEMDOS    equ       1                   Trap-Nummer

*     Einfache lesende Funktionen ****

*     conin wartet und liest dann ein Zeichen von der Tastatur
*     und echot es auf den Bildschirm, das ASCII-Byte ist in D0.B
*     Im Low Byte des hoeherwertigen Wortes von D0.L ist der Scan-Code

conin     move.w    #1,-(sp)            Funktionsnummer
          trap      #GEMDOS             Aufruf
          addq.l    #2,sp               Stack berichtigen
          rts                           Jetzt D0.L oder D0.B auswerten

*         auxin wartet und holt ein Zeichen von der RS232-Schnittstelle

auxin     move.w    #3,-(sp)            Funktions-Nr.
          trap      #GEMDOS
          addq.l    #2,sp               Stack berichtigen
          rts                           ASCII im Register D0.B

*     conio ist eine gemischte E/A-Funktion fuer Tastatur und Bildschirm
*     Mit D0.W=$ff erhaelt man in D0.L eine 00. wenn keine Taste betaetigt,
*     bei betaetigter Taste holt man ein Zeichen wie bei conin.
*     Der Aufruf ist hierfuer conioin.
*     Mit D0.W<>$ff schreibt man das Zeichen auf den Bildschirm,
*     der Aufruf hierfuer ist conio, wobei das Zeichen in D0.W sein muss.

conioin   move.w    #$ff,d0             Fuer Zeichenpruefung/Empfang
conio     move.w    d0,-(sp)            Zeichen in d0
          move.w    #6,-(sp)            Funktions-Nr
          trap      #GEMDOS             Aufruf
          addq.l    #4,sp
          tst.w     d0                  Eingabe gleich pruefen
          rts

*     dircon liest ein Zeichen von der Tastatur wie conio,
*     ohne Echo-Ausgabe auf den Bildschirm,
*     Control-Zeichen werden durchgelassen

dircon    move.w    #7,-(sp)            Funktionsnummer
dircon1   trap      #GEMDOS
          addq.l    #2,sp
          rts                           Nun Zeichen in D0 auswerten

*     dirconctrl liest ein Zeichen wie dircon, es werden aber
*     CTRL-Zeichen durchgelassen, die sich am BS auswirken

dirconctrl move.w   #8,-(sp)
          bra.s     dircon1

*     readline liest einen mit CR abgeschlossenen String
*     von der Tastatur in einen Buffer, dessen Adresse beim
*     Aufruf in D0 stehen muss. Das erste Byte im Puffer muss die
*     maximale Laenge des Datenteils enthalten, das zweite enthaelt nach
*     rts die wirkliche Laenge. Im 3. Byte beginnt der Datenteil des Puffers.
*     CTRL-Tasten wirken sich beim Editieren aus
```

Fortsetzung

GEMDOS-Funktionen zur Einbindung (Fortsetzung)

```
readline  move.l    d0,-(sp)          Zeiger auf den Buffer in d0
          move.w    #10,-(sp)         Funktions-Nr
          trap      #GEMDOS
          addq.l    #6,sp             Stack berichtigen
          rts

*         coninstat prueft, ob ein Zeichen im Tastaturbuffer vorliegt.
*         D0.W ist 00, wenn nicht, D0.W ist negativ, wenn ja

coninstat move.w    #11,-(sp)         Funktionsnummer
coninstl  trap      #GEMDOS
          addq.l    #2,sp
          tst.w     d0                Status gleich pruefen
          rts

*    auxinstat prueft, ob ein Zeichen am RS232-Eingang vorliegt
*    D0.W wird gesetzt, wie bei coninstat
*
auxinstat move.w    #18,-(sp)         Funktionsnummer
          bra.s     coninstl

*    Einfache schreibende Funktionen *****

*    conaut gibt ein in D0.W, Low Byte uebergebenes Zeichen auf
*    die Standardausgabe (Bildschirm) aus, das High Byte sollte 00 sein.

conout    move.w    d0,-(sp)          Zeichen auf den Stack
          move.w    #2,-(sp)          Funktions-Nr.
conoutl   trap      #GEMDOS           Aufruf
          addq.l    #4,sp
          rts
*
*    auxout gibt gibt ein Zeichen in D0.W auf den RS232-Kanal aus,
*    Bedingungen sonst wie bei conout
*
auxout    move.w    d0,-(sp)
          move.w    #4,-(sp)
          bra.s     conoutl
*
*    prtout gibt ein Zeichen in D0.W auf den Printerport (Centronics) aus,
*    Bedingungen sonst wie bei conout
*
prtout    move.w    d0,-(sp)
          move.w    #5,-(sp)
          bra.s     conoutl
*
*    printline gibt einen String Byte fuer Byte auf die Standard-Ausgabe
*    aus, bis eine 00 als Begrenzer gefunden wird.
*    In D0.L ist die Adresse des Stringbeginnes zu uebergeben

printline move.l    d0,-(sp)          Stringadresse
          move.w    #9,-(sp)     Funktion
          trap      #GEMDOS           Aufruf
          addq.l    #6,sp             Stack berichtigen
          rts                         Gffs. CR/LF nachschieben

*
*    crlf gibt ein CR und LF auf die Standard-Ausgabe (Bildschirm)

crlf      move.w    #$0d,d0           Das CR
          bsr.s     conout
          move.w    #$0a,d0           Das LF
          bsr.s     conout
          rts
```

Fortsetzung

GEMDOS-Funktionen zur Einbindung (Fortsetzung)

```
*     conoutstat prueft, ob die Standard-Ausgabe (Bildschirm)
*     empfangsbereit ist. Dann ist D0.W=$ff, sonst 00.

conoutstat move.w   #16,-(sp)
conoutssl  trap     #GEMDOS
           addq.l   #2,sp
           tst.w    d0                    Status gleich pruefen
           rts

*     printstat prueft, ob die Centronics-Schnittstelle bereit ist,
*     sonst wie conoutstat

printstat  move.w   #17,-(sp)
           bra.s    conoutssl

*     auxoutstat prueft die RS232-Ausgabe wie printstat

auxoutstat move.w   #19,-(sp)
           bra.s    conoutssl

*     Routinen zum Abbruch des Prozesses

*     term kehrt zum aufrufenden Prozess zurueck
*     Aufzurufen mit jmp term

term       clr.w    -(sp)                 Funktions-Nummer 00
           trap     #GEMDOS
```

Kapitel 29

Das erweiterte XBIOS

Mit Trap #14 werden etwa 40 Funktionen des erweiterten BIOS erreicht. Die Übergabeprozedur auf dem Stack, der Funktionsaufruf mit Nummer und die anschließend notwendige Berichtigung des Stackpointers verlaufen wie bei den BIOS-Funktionen mit TRAP #13 oder wie bei GEMDOS-Aufrufen.

Viele Funktionen sind auf die spezielle Hardware des Atari zugeschnitten, ohne deren Dokumentation auch eine langatmige Beschreibung ihrer Parameter und ihrer Wirkung wenig sinnvoll ist. In solchen Fällen wird die Funktion zwar meistens erwähnt, der Leser sollte bei besonderem Interesse jedoch beim Händler die Dokumentation zum Entwicklungspaket des Atari studieren. Offensichtlich wertlose Funktionen werden nicht aufgeführt.

XBIOS-Funktionen, mit TRAP #14 aufzurufen

```
*    XBIOS-Funktionen, mit TRAP #14 aufzurufen

*    Video-Verwaltung
INITMOUSE   equ  0        Maus initialisieren oder abschalten
SSBRK       equ  1        Speicherreservierung beim Booten, sonst wertlos
PHYSBASE    equ  2        Beginnadresse Video-RAM holen
LOGBASE     equ  3        Logische Adresse Bildschirmbasis holen
GETREZ      equ  4        Aktuelle Bildschirmaufloesung holen
SETSCREEN   equ  5        Bildschirmsteuerung/Basisadressen aendern
SETPALETTE  equ  6        Laden einer ganzen neuen Farbpalette
SETCOLOR    equ  7        Setzen/Pruefen einer Farbe in der Palette

*    E/A-Funktionen
IOREC       equ  14       Buffer-Zeiger serieller Eingabegeraete holen
RSCONF      equ  15       RS232-Schnittstelle konfigurieren
KEYTBL      equ  16       Zeiger auf Tabelle der Tastaturdecodierung umsetzen
SCRDMP      equ  20       Hardcopy des Bildschirmes auf Drucker
CURSCONF    equ  21       Cursor konfigurieren/abschalten
BIOSKEYS    equ  24       Original-Tastaturbelegung wiederherstellen
IKWDWS      equ  25       Befehlsstring an das intelligente Keyboard senden
SETPRT      equ  33       Drucker-Konfiguration lesen/aendern
KBDVBASE    equ  34       Zeiger holen, der auf einen Adressenblock von
*                         7 E/A-Routinen zeigt
KBRATE      equ  35       Lesen/Setzen des Tastatur-Repeats
PRTBLK      equ  36       Unterprogramm zu Funktion 20, Hardcopy

OFFGIBIT    equ  28       1 Bit in Port A des Sound-Chips setzen fuer Funk-
*                         tionen der Floppy/RS232/Centronics
ONGIBIT     equ  30       1 Bit gezielt loeschen, wie vor

*    Floppy-Ansprache
FLOPRD      equ  8        Einen oder mehrere Sektoren von Floppy lesen
FLOPWR      equ  9        Sektor(en) auf Floppy schreiben
FLOPFMT     equ  10       Eine Spur auf Floppy formatieren
PROTOBT     equ  18       Bootsektor fuer Diskette definieren
FLOPVER     equ  19       Verify fuer einen oder mehrere Floppy-Sektoren

*    Interruptverwaltung
MFPINT      equ  13       Interrupt-Vektor fuer 68901 MFP-Baustein setzen
JDISINT     equ  26       Einen Interrupt vom 68901 sperren
JENABINT    equ  27       Einen Interrupt vom 68901 freigeben
XBTIMER     equ  31       Einen Timer im 68901 starten
VSYNC       equ  37       Auf Interrupt des Bildwechsels warten

*    Verschiedene Funktionen
RANDOM      equ  17       Zufallszahl in 24 Bit holen, mit fuehrenden Nullen
SUPEXEC     equ  38       Routine ab Adresse im Supervisor-Modus ausfuehren
PUNTAES     equ  39       AES abschalten und ggfs. RAM freigeben

*    Sound-Funktionen
MIDIWS      equ  12       String an MIDI-Port ausgeben
GIACCESS    equ  28       Lesen/Schreiben eines Registers im Sound-Chip
DOSOUND     equ  32       Aufruf einer Befehlsfolge fuer den Sound-Chip
```

Nebenstehend sind die wichtigsten Funktionen des XBIOS gruppenweise aufgeführt. Dabei wurde folgende Einteilung gewählt:

- Video-Verwaltung
- E/A-Funktionen
- Floppy-Ansprache
- Interruptverwaltung
- Verschiedene Funktionen
- Sound-Funktionen

Die Bezeichnung **XBIOS** steht für E**x**tended **BIOS**. Es handelt sich um Funktionsaufrufe, die die besondere Hardware des Atari auf einer physikalischen Ebene zu hantieren gestatten, ähnlich, wie es das BIOS für die Grundfunktionen der E/A erlaubt. Die Routinen werden vor allem vom System selbst benutzt. So ist es beispielsweise für ein Anwenderprogramm ziemlich wertlos, eine Floppy mit der Funktion Nr. 10 zu formatieren. Das besorgt man normalerweise vom Desktop her, ebenso die Einstellung des Tastaturrepeats, der Druckerkonfiguration usw. Auch die Tastaturbelegung wird man im allgemeinen nicht ändern. Im weiteren Verlauf des Buches werden wir daher nur in besonderen Fällen auf einzelne Funktionen des XBIOS eingehen, weil es nicht das Ziel sein kann, in einem Buch zur Assemblerprogrammierung sämtliche Funktionen im Atari zu belegen. Unser Ziel ist es vor allem, den Gebrauch der 68000-Befehle auf diesem Rechner zu zeigen, und das vor allem in einer zeichen-orientierten Umgebung, nicht in der Grafik.

Kapitel 30

Benutzung des Toolkits

Die Benutzung unseres GEMDOS-Toolkits wird erstmalig geübt. Es soll eine Zeile Text nach interaktiver Aufforderung entgegengenommen, gespeichert und dann wieder ausgegeben werden.

Der Quelltext des nachfolgenden Programmes ist wiederum für den GST-Assembler geschrieben. Bei Benutzung eines anderen Assemblers, z. B. von DRI, muß man kleine Änderungen vornehmen, auf die im Text hingewiesen wurde.

Im Programm sollen die GEMDOS-Funktionen READLINE und PRINTLINE erprobt werden, die einen String von der Tastatur in einen Buffer lesen bzw. einen String aus einem Buffer ausgeben. READLINE verlangt im 1. Byte des Buffers eine Angabe für die maximale Länge des Eingabestrings. Wir definieren diese Länge in der Befehlszeile bei MAIN zu 40, um später schnell eine Reaktion zu sehen, wenn der Buffer voll ist (READLINE bricht dann die Eingabe spontan ab). In den drei Zeilen ab MAIN1 wird ein als Konstante und mit 00 begrenzter gespeicherter Text zusammen mit CRLF auf den Bildschirm geschrieben. Bei READ wird der Zeiger auf den gewählten Eingabe-Buffer in D0 übergeben. Danach wird eine Zeile eingelesen, bis entweder ein CR auftritt oder bis der Buffer voll ist.

Der empfangene String ist dann mit einem CR begrenzt, wir benötigen für PRINTLINE jedoch eine 00 als Begrenzer. Die Fra-

ge ist, wie wir die notwendige Stelle der Null innerhalb des Buffers adressieren sollen. Nun ist zu erinnern, daß READLINE im 2. Byte des Buffers abliefert, wie lang der String geworden ist. Beim Label PRINT0 laden wir daher zunächst das Adreßregister A1 mit dem Zeiger auf BUFFER als Basisadresse. Bei PRINT1 wird dann indirekt mit einem Offset von +1 auf das Byte mit der Längenangabe adressiert. Diese steht nun im Register D0. Und nun kommt bei PRINT2 eine indirekte Adressierung mit Offset und Index in den Buffer hinein, dorthin, wo das CR steht und wo eine Null hin muß. Wir betrachten die Bildung der Zieladresse: "2(A1" zeigt auf die Stelle in BUFFER, wo der Nutz-String beginnt. Das ist 2 Bytes hinter der Verwaltungs-Information. Der Adressierungsbestandteil ",D0.W)" bedeutet, daß der Inhalt des Registers D0 als ein Index oder als ein weiterer Offset benutzt wird. Dieser Offset ist die erreichte Länge des Strings, die zu "2(A1" als effektive Adresse hinzuaddiert wird. – In den folgenden Programmzeilen sind wir nun in der Lage, den empfangenen String per PRINTLINE zusammen mit den notwendigen Zeilentransporten wieder auszusenden.

Mit BRA.S MAIN1 gelangen wir an den Programmbeginn zurück. – Bei der Erprobung des Programmes werden wir feststellen, daß wir keine eigene Abfrage auf ein empfangenes Kontrollzeichen CTRL-C einbauen müssen, um den Prozeß abzubrechen. Diese Abfrage ist bereits in READLINE enthalten. Mit CTRL-C kommen wir zum TOS zurück.

Am Schluß des Programmes finden wir noch folgende Anweisungen: 'NOLIST' und 'INCLUDE GEMDOSFN' (das hier benutzte File GEMDOSFN soll einbezogen, aber nicht gelistet werden), dann wieder LIST und Mit DC.B und DS.B die Anlage einer Textkonstanten und die Reservierung von Speicherraum für den BUFFER.

Die Logik des Programmes ist bei der Benutzung der vorformulierten GEMDOS-Routinen denkbar einfach. Mit wenigen Zeilen kommen wir in die Anfänge eines Textsystems. – Bei der Erprobung und nachdem wir das Programm für TOS angemeldet

haben, machen wir nun einige Beobachtungen: Beim Beginn und immer, wenn wir eine Zeile eingegeben haben, erhalten wir den textlichen Prompt "Bitte eine Zeile eingeben". Das ist wie erwartet. Wenn wir aber nun Tasten mit den deutschen Sonderzeichen drücken, so erscheinen im Echo des Eingabestrings ungewöhnliche Sonderzeichen, der Ausgabestring von PRINTLINE hat jedoch die richtige Wiedergabe. CTRL-Zeichen erzeugen ferner einen Hochpfeil und einen Klammeraffen. Das READLINE des GEMDOS ist damit wenig geeignet, um in Absicherung gegen Fehlbedienungen vernünftigen deutschen Eingabetext anzuzeigen. Für solche Anwendungen müssen wir auf einfachere BIOS-Routinen zurückgehen, dabei die Zeichen filtern und ggfs. umcodieren.

Erste Benutzung von GEMDOS-Funktionen

```
* Erste Benutzung von GEMDOS-Funktionen
* Eingabe eines Strings mit nachfolgender Wiederausgabe
* Programmabbruch mit CONTRL-C

  SECTION ONE
* text     fuer DRI-Assembler
* _main    dito als 1. Label

main       move.b    #40,buffer      Definiere max. Zeilenlaenge
mainl      move.l    #text,d0        Zeiger auf konstanten Text setzen
           jsr       printline       Gib den Text aus per GEMDOS
           jsr       crlf            Zeilenvorschub per GEMDOS

read       move.l    #buffer,d0      Zeiger auf den Buffer als Parameter
           jsr       readline        String holen per GEMDOS

           move.l    a1,-(sp)        Adressregister sichern
print0     movea.l   #buffer,a1      Bufferadresse nach a1
           clr.w     d0              Moegliche Vorzeichenerweiterung bedenken
print1     move.b    1(a1),d0        In Buffer+1 steht die erreichte Laenge
print2     move.b    #0,2(a1,d0.w)   Begrenzer 00 einsetzen
           move.l    (sp)+,a1        Register zurueckladen
           jsr       crlf            In neue Zeile gehen

           move.l    #buffer+2,d0    Zeige auf Stringbeginn
           jsr       printline       Textausgabe per GEMDOS
           jsr       crlf            Neue Zeile
           bra.s     mainl           Weitermachen
 NOLIST
 INCLUDE   GEMDOSFN

 LIST
*    data              Fuer DRI-Assembler
text       dc.b      'Bitte eine Zeile eingeben',00 ;Begrenzer 0
*    bss               Fuer DRI-Assembler
buffer     ds.b      256             Eingabe-Buffer

 END
```

MIT TEXTEN ARBEITEN

MIT TEXTEN ARBEITEN

Kapitel 31

Fallentscheidungen

Wo in einem Programm von einem Punkt aus viele Verzweigungen möglich sind, da sollte man die notwendigen Abfragen nicht durch eine Aneinanderreihung von mehreren Vergleichsbefehlen bewirken, sondern eine Entschlüsselungsschleife nebst Sprungverteiler benutzen, der einen übersichtlichen Programmaufbau gewährleistet. Dazu hier ein Vorschlag, der das Schleifenkonstrukt *Decrement* und *Branch* benutzt und der indirekte Sprünge über den Inhalt eines Adreßregisters implementiert.

Reihenabfragen mit einer Entschlüsselung kommen in der Programmierung häufig vor. Man denke an die Befehlsebenen eines Monitor- oder Editorprogrammes, wo mit einem Tastendruck eine Funktion ausgelöst wird. Oder man denke an den Fall, wo Schlüsselworte in einer Eingabezeile zu gezielten Aktionen führen sollen. In höheren Programmiersprachen nennt man solche Konstrukte auch ON ... GOTO oder CASE und OTHERWISE. Auch in der Assemblersprache sind solche Entschlüsselungen mit einfachen Mitteln zu bewirken. Dazu das Beispielprogramm FALLENT.

Zur Verschaulichung des Prinzips stellen wir uns die Aufgabe, daß die vier Eingabetasten großes A ... D einen Text 'Taste A' usw. zur Ausgabe bringen sollen. Alle übrigen Tasten führen zur Textausgabe 'Unzulaessige Taste'. Das ist dann gewissermaßen der Fall OTHERWISE.

Nach der Meldung des Programmes wird beim Label MAIN3 ein Tastendruck eingeholt und die Register A0 und D1 dann vorübergehend auf dem Stack gesichert. Nun kann A0 mit LEA.L mit der Basis der Tastaturtabelle KEYTABEL geladen werden. Das in MAINLOOP benutzte Schleifenkonstrukt Decrement und Branch setzt voraus, daß der Schleifenzähler in einem Datenregister geführt wird. Wir nehmen hier D1, das für 4 Durchläufe mit dem Wert 3 vorgeladen wird. Mit dem Befehl DBEQ D1,MAINLOOP wird die Schleife verlassen, wenn das Eingabezeichen in D0 mit einem Eintrag in der Tabelle KEYTABEL übereinstimmt oder wenn D1 negativ geworden ist. Der entscheidende Vergleichsbefehl in dieser Schleife ist CMP.B (A0,D1.W),D0. Hier ist der Quelloperand einmal wieder 'indirekt mit Index D1' adressiert, womit wir rückwärtsschreitend die KEYTABEL überstreichen.

Beim Verlassen der MAINLOOP haben wir einen erlaubten Tastendruck gefunden, wenn Register D1 noch positiv ist. Also wird der Befehl TST.W D1 zu dieser Feststellung gegeben. In diesem Falle kommen wir nach MAIN5 (sonst sofort nach MAIN4). Hier müssen wir etwas über die möglichen Adressierungsarten nachdenken, um schließlich zu den exekutierenden Routinen ROUTA ... ROUTD zu gelangen, die als Programmstummel ganz einfach nur meldende Texte ausgeben sollen. Bei MAIN5 laden wir A0 zunächst mit der Vektor-Basisadresse von ROUTINEN. Je nach vorliegendem Fall A ... D müssen wir 0, 4, 8 oder 12 Bytes rechts davon ansetzen. Wir laden daher Register D0 mit dem laufenden Offset, der einen dieser Werte hat. Nun erst können wir A0 mit dem Inhalt von einem der Vektoren laden (MOVE.L(A0,D0.L),A0). JMP (A0) führt danach zu ROUTA oder ROUTB usw. und dann weiter zu MAIN4, wo die entsprechende Textausgabe erfolgt.

Die Konstruktion ist bei der Aufschlüsselung etwas umständlich, weil der MC68000 noch keine indirekten Adressierungen über Speicheradressen erlaubt, sondern nur solche über Adreßregister. Gleichwohl sind solche Sprungverteiler vor allem dann interessant, wenn viele mögliche Verzweigungen in Betracht

kommen, die zu klar strukturierten Programmabschnitten weiterführen sollen.

FALLENT Fallentscheidungen

```
* FALLENT Fallentscheidungen
* Programmabbruch mit CTRL-C

 SECTION  ONE
main      bsr       crlf            Neue Zeile
          move.l    #text1,d0       Zeiger auf interaktiven Text setzen
          jsr       printline       Ausgabe per GEMDOS
main2     jsr       dircon          Zeichen holen per GEMDOS
          cmpi.b    #$03,d0         Ist es CTRL-C ?
          beq       abbruch         ja
          movem.l   a0/d1,-(sp)     2 Register sichern
          move.l    #4*4,offset     Hinter letztes der 4 Items zielen
          moveq.l   #3,d1           Zaehler einrichten fuer 4 Items
main3     lea.l     keytabel,a0     Zeiger auf erlaubte Befehlstasten
mainloop  subq.l    #4,offset       Offset laufend machen
          cmp.b     (a0,d1.w),d0    Vergleich Taste mit Tabelle keytabel
          dbeq      d1,mainloop     Verlasse, wenn gleich, sonst mainloop
          tst.w     d1              d1 unter Null?
          bpl.s     main5           nein, es wurde etwas gefunden
          move.l    #text2,d0       Ungueltigen Befehl anzeigen
main4     movem.l   (sp)+,a0/d1     Register zurueck
          jsr       printline       Textzeile per GEMDOS
          bra.s     main            Neuer Durchlauf

main5     move.l    #routinen,a0    Basisadresse Sprungverteiler
          move.l    offset,d0       Korrektur fuer das gefundene Item
          move.l    (a0,d0.l),a0    Inhalt unter Adresse laden
          jmp       (a0)            Fuehre Routine des Falles aus

routa     move.l    #texta,d0       Fuer Taste A
          bra.s     main4
routb     move.l    #textb,d0       Fall B
          bra.s     main4
routc     move.l    #textc,d0       Fall C
          bra.s     main4
routd     move.l    #textd,d0       Fall D
          bra.s     main4

abbruch   jmp       term            Programm per GEMDOS abbrechen

routinen  dc.l      routa,routb,routc,routd ;Sprungadressen
text1     dc.b      'Bettigen Sie eine der Tasten ABCD   ',0
text2     dc.b      'Unzulaessige Taste',0
keytabel  dc.b      'ABCD'

* Antwort-Texte
texta     dc.b      'Taste A',0
textb     dc.b      'Taste B',0
textc     dc.b      'Taste C',0
textd     dc.b      'Taste D',0

* Arbeitsspeicher
offset    ds.l      1               Zaehler fuer Offset zu a0

 NOLIST
 INCLUDE  GEMDOSFN                  Siehe GEMDOS Toolkit
 END
```

Kapitel 32

Filter

Die nachstehenden Unterprogramme überprüfen ein im Register D0 als Byte übergebenes ASCII-Zeichen auf seine Zugehörigkeit zu verschieden gebildeten Mengen, als da sind: Hexziffern, Buchstaben, alphanumerische Zeichen (Buchstaben und Ziffern), Dezimalziffern, druckbare Zeichen. Filter für so gebildete Wertebereiche, technisch gesprochen: Bandpässe kommen in jeder besseren Programmierung vor, gleichgültig, in welcher Sprache der Ausdruck gewählt wird.

Es handelt sich hier um ganz einfache Algorithmen unter Anwendung des Vergleichsbefehles CMPI, bei dem eine gedachte Subtraktion 'Zieloperand minus Quelloperand' (hier: Direktwert) stattfindet und bei dem das Bedingungscoderegister entsprechend gesetzt wird. Im Gegensatz zu anderen CPU-Typen, z. B. 6502, wird bei einem "Borgen" (wenn der Zieloperand kleiner als der Vergleichsoperand ist) das Carry-Flag gesetzt. Ist er jedoch größer oder gleich, so wird das Carry-Flag gelöscht.

Die schon durch die Kommentare in der nachstehenden Liste ausreichend erklärten Module werden hier "auf Vorrat" abgedruckt. In der einen oder anderen Form wird ihr Quelltext-File FILTER bei späteren Assemblierungen mit INCLUDE herangezogen und als bekannt vorausgesetzt. Zum Ablauf ist zu bemerken, daß das im Register D0.B übergebene Byte nicht verändert wird. Weiterhin: Alle Unterprogramme werden mit einem definierten Status im N-Flag und im Z-Flag verlassen. Gehört das

Zeichen zur gesuchten Zeichenmenge, so ist das Ergebnis positiv (Verzweigung mit BPL oder BMI nach dem Aufruf) und außerdem ist das Z-Flag gesetzt (BEQ, wenn zugehörig).

Gleichartige Filter lassen sich mühelos auch unter Einbeziehung der deutschen Sonderzeichen bei den Buchstaben und z. B. für Trennzeichen in Kommandozeilen oder in anderen Eingabetexten schreiben.

FILTER Eine Reihe von Filter-Routinen

```
* FILTER Eine Reihe von Filter-Routinen

* HEXFILT  zeigt in filtflag an, ob ein Zeichen in Gross-
* oder Kleinschreibung eine Hexziffer ist
hexfilt   cmpi.b    #'g',d0         Kleines g und folgende Zeichen
          bcc.s     nothex          sind nicht hex
          cmpi.b    #'a',d0         a ... f
          bcc.s     ishex           sind zugelassen
          cmpi.b    #'G',d0         Grosses G ...
          bcc.s     nothex          sind nicht hex
          cmpi.b    #'A',d0         A ... F, Grossbuchstaben
          bcc.s     ishex           wieder zugelassen
          cmpi.b    #':',d0         Dazwischen liegen wieder keine
          bcc.s     nothex          Hexziffern
          cmpi.b    #'0',d0         Ziffernbereich 0 ... 9 erlaubt
          bcs.s     nothex          Zeichen liegt darunter
ishex     clr.w     filtflag        =00
          rts
nothex    move.w    #$ffff,filtflag Flag wird negativ
          rts

* ALPHA ist ein Filter fuer Buchstaben A...Z und a...z
alpha     cmpi.b    #$7b,d0         Groesser 'z'?
          bcc.s     notalpha        ja
          cmpi.b    #'a',d0         Buchstaben 'a'...'z'?
          bcc.s     isalpha         ja
          cmpi.b    #$5b,d0         Groesser 'Z'?
          bcc.s     notalpha        ja, Sonderzeichen
          cmpi.b    #'A',d0         Buchstaben 'A'...'Z'?
          bcs.s     notalpha        nein, kleiner
          clr.w     filtflag
isalpha   rts                       FILTFLAG=0, Zeichen ist alpha
notalpha  move.w    #$ffff,filtflag
          rts                       FILTFLAG = negativ

* ALPHNUM ist ein Filter fuer Ziffern 0...9 und Buchstaben wie bei ALPHA
alphnum   cmpi.b    #$3a,d0         Groesser '9'?
          bcs.s     alpha           ja, keine Ziffer
          cmpi.b    #'0',d0         Die '0' gehoert dazu
          bcs.s     notalpha        ASCII-Wert ist geringer
          clr.w     filtflag
          rts                       Filtflag=0, Zeichen ist alphanumerisch

* DEZZIF ist ein Filter fuer Dezimalziffern 0...9
dezzif    cmpi.b    #$3a,d0         Groesser als '9'?
          bcc.s     notalpha        ja
          cmpi.b    #'0',d0         Unter '0'?
          bcs.s     notalpha        ja, FILTFLAG=negativ
          clr.w     filtflag        FILTFLAG=0
          rts

* DRUCKBAR ist ein Filter fuer die Zeichen Space...Tilde ($7e)
druckbar  cmpi.b    #$7f,d0         Rubout oder groesser?
          bcc.s     notalpha        ja, nicht druckbar
          cmpi.b    #' ',d0         Space?
          bcs.s     notalpha        kleiner, nicht druckbar
          clr.w     filtflag        =00
          rts

filtflag  ds.w      1               Flag-Variable
```

Kapitel 33

Ein Editor

Ziel dieses Abschnittes ist es, vor allem das Unterprogramm **GETLINE** zur Verfügung zu stellen, mit dem eine Eingabezeile in einen beliebig zu bestimmenden Eingabepuffer eingelesen werden kann. Eingaben dieser Art werden wir bei künftigen Programmierungen häufiger benötigen. Mit dieser Dienstleistung unter GEMDOS in der Hand, können die Anfänge eines Textsystemes aufgezeigt werden.

Im Abschnitt "Benutzung des Toolkits" haben wir festgestellt, daß die GEMDOS-Funktion **READLINE** bei der Verwendung von Sonderzeichen Eigenheiten aufweist, die sie für eine narrensichere zeilenweise Texteingabe ziemlich ungeeignet macht. Es wurden daher zunächst die Programmteile FILTER und das Unterprogramm **GETLINE** geschrieben, um in der Eingabe mit DIRCON einen Zeichenstring aus nur druckbaren Zeichen zu erhalten, der identisch im Tastaturpuffer BUFFER abgelegt werden kann. GETLINE läßt neben der Eingabe und Speicherung solcher Zeichen nur noch folgende Besonderheiten zu: Die Taste ESCAPE führt zum Abbruch der Texteingabe. Ein CR schließt eine Zeile ab und führt zur Programmebene des Aufrufes zurück. Ein Entweichen des Cursors aus der Eingabezeile wurde unmöglich gemacht. Die Taste Backspace verlangt besondere Aufmerksamkeit: Backspace setzt mit CONOUT den Cursor nach links, läßt aber das zu löschende Zeichen zunächst noch unter dem Cursor sichtbar erscheinen. Es muß durch ein Leerzeichen (Blank) überschrieben werden. Wenn man das ausführt, dann

sitzt der Cursor jedoch wieder rechts von der eigentlich beabsichtigten Schreibstelle. Daher ist in GETLINE nach der Überschreibung des zu löschenden Zeichens ein weiteres Backspace an den Bildschirm auszugeben. Gleichzeitig müssen der Zeiger in den Tastaturpuffer (Register A5) und der mitlaufende Zähler (A4) berichtigt werden.

Mit der Funktion GETLINE kann man bereits die Anfänge eines Textsystems aufbauen, was wir hier auch gleich bei passender Gelegenheit einbinden: Das Hauptprogramm holt nach dem Label MAIN1 mit JSR Getline jeweils eine Zeile in den Tastaturbuffer und transportiert sie ab Label MAIN2 jeweils an die Einsatzstelle am Ende des bisher schon benutzten Textspeichers (der Zeiger ist in der Variablen EDIPNTR gespeichert).

Wurde die Texteingabe in GETLINE mit ESCAPE abgeschlossen, so gelangt man in eine einfache Kommando-Ebene MAINCMD, in der man mit Tastendruck das Programm beenden kann (Taste CTRL-C), weitere Texteingaben vornehmen ('a' wie Append), auf die oberste Zeile positionieren (Taste 't') oder eine Zeile weitersetzen kann (Taste 'd' wie Down). – Es ist klar, daß ein Textsystem wesentlich mehr leisten und insbesondere die Massenspeicherung lösen muß, die wir in einem weiteren GEMDOS-Abschnitt behandeln. – Wenn man bereits einen Editor nebst Assembler und Linker zur Verfügung hat, so kann man sich mit den dargestellten Funktionen bereits einen Editor nach dem eigenen Geschmack schreiben und letztlich sogar auch ein Datenbanksystem mit Suchfunktionen. Die Grundfunktion dafür ist GETLINE. Bemerkenswert ist, das alles mit wenigen Zeilen Quelltext erledigt werden kann.

GETLINE

```
* GETLINE
* Eingabe einer Zeile bis CR
* Programmabbruch mit CTRL-C

 SECTION  ONE
main      move.l    #text1,d0        Zeiger auf interaktiven Text setzen
          jsr       printline        Ausgabe per GEMDOS
          movem.l   a3-a5,regsave    Register sichern
main0     lea       editor,a3        Zeige auf Beginn des Textspeichers
          move.l    a3,edipntr       Einsatzpunkt merken
          clr.b     (a3)             Dort eine Null als Begrenzer ablegen
main1     jsr       crlf             Neue Zeile
          lea       buffer,a5        Benutze BUFFER
          jsr       getline          1 Zeile in den Buffer lesen
          cmpi.b    #$1b,d0          lag ESCAPE vor?
          beq.s     maincmd          ja, nun Kommando holen
          lea       buffer,a5        Zeilentransport vorbereiten
main2     move.b    (a5)+,(a3)+      Vom Buffer in den Editor
          bne.s     main2            Begrenzer 00 noch nicht angetroffen
          move.b    #$0d,-1(a3)      00 mit CR ueberschreiben
          clr.b     (a3)+            00 als Begrenzer anhaengen
          move.l    a3,edipntr       Einsatzpunkt merken
          bra.s     main1            Weitere Zeile holen

maincmd   jsr       crlf             Neue Zeile
          move.l    #text2,d0        Befehlsebene anzeigen
          jsr       printline
maincmd1  jsr       crlf
          jsr       dircon           Antwort-Taste holen
          cmpi.b    #'a',d0          Append Text ans Ende?
          beq.s     main1            ja
          cmpi.b    #$03,d0          CTRL-C als Abbruch?
          beq.s     abbruch          ja
          cmpi.b    #'t',d0          Auf Topzeile setzen?
          beq.s     topzeile         ja
          cmpi.b    #'d',d0          Down, in Folgezeile setzen?
          beq.s     down             ja
          bra.s     maincmd1         Kein gueltiger Befehl
abbruch   movem.l   regsave,a3-a5    Register zurueckholen
          jmp       term
topzeile  lea.l     editor,a3        Zeiger auf Textbeginn
down      tst.b     (a3)             Sind wir am Ende des Editors?
          beq.s     maincmd          ja, neuen Befehl holen
          jsr       showline         Zeile anzeigen
          bra.s     maincmd          Weiteren Befehl mit Prompt
```

Fortsetzung

GETLINE (Fortsetzung)

```
getline   suba.l   a4,a4          Setze zu 00
getline2  jsr      dircon         Zeichen holen
          cmpi.b   #$0d,d0        Zeilenabschluss?
          beq.s    getlincr       ja
          cmpi.b   #$1b,d0        ESCAPE = Abbruch der Eingabe?
          beq.s    getlinez       ja
          cmpi.b   #$08,d0        Backspace?
          bne.s    getline3       nein
          cmpa.l   #0,a4          Sind wir in 1. Spalte?
          beq.s    getline2       ja, dann Backspace ignorieren
          jsr      conout         Zeichen ausgeben mit GEMDOS
          move.b   #' ',d0        Altes Zeichen mit Space
          jsr      conout         ueberschreiben
          move.b   #$08,d0        Cursor wieder auf alte
          jsr      conout         Schreibstelle setzen
          suba.l   #1,a5          Pointer berichtigen
          suba.l   #1,a4          Ebenso Zaehler
          bra.s    getline2
getline3  jsr      druckbar       Filtern lt. INCLUDE FILTER
          bmi.s    getline2       Nicht druckbar
          move.b   d0,(a5)+       Abspeichern in den Buffer
          jsr      conout         Und Ausgabe
          adda.l   #1,a4          Mitlaufender Zaehler +1
          cmpa.l   #80,a4         Zeile voll?
          bcc.s    getlincr       ja, 80 Zeichen, Zwangsabbruch
          bra.s    getline2
getlincr  clr.b    (a5)           00 als Begrenzer in den Buffer
getlinez  rts

showline  move.b   (a3)+,d0       Laufendes Byte einer Zeile
          beq.s    showlret       Abbruch, wenn 00
          cmpi.b   #$0d,d0        War letztes Byte CR?
          beq.s    showlcr        ja, return
          jsr      conout         Ausgabe
          bra.s    showline       Folgezeichen
showlcr   adda.l   #1,a3          Hinter das CR setzen
showlret  rts

 NOLIST
 INCLUDE  GEMDOSFN                Einzubindende Quellfiles,
 INCLUDE  FILTER                  siehe fruehere Abschnitte
 LIST
textl     dc.b     'Eingabe von Text',0
text2     dc.b     'Befehl eingeben: a t d CTRL-C'

* Arbeitsspeicher
regsave   ds.l     3              Register-Ablage zur Sicherung
edipntr   ds.l     1              Zeiger auf Einsatzpunkt Editor
buffer    ds.b     100            Tastatur-Eingabe-Puffer
editor    ds.b     128*1024
ediende                           ;Label der Bereichsbegrenzung
 END
```

Kapitel 34

Ein Parser

In vielen Programmen findet ein Vergleich von Textketten statt: Eingaben sollen gegen eine Tabelle von Schlüsselworten verglichen werden, um z. B. eine Aktion auszulösen. Man denke da an die Entschlüsselung einer Kommandozeile oder an einen BASIC-Interpreter im Tischrechner-Modus, der Befehlsworte in den Aufruf von Programmteilen umsetzt. Vergleiche dieser Art werden auch bei der Übersetzung von Quelltexten in Assembler oder anderen höheren Sprachen sowie bei der Suche in Datenbanken durchgeführt. – Das nachfolgende Beispiel enthält eine ganz einfache Entschlüsselung, man nennt sie auch "Parser", für solche Aufgaben. Dabei werden häufig die Adressierungsarten "Adreßregister indirekt mit Offset" und "Adreßregister indirekt mit Postinkrement" eingesetzt.

Unser Parser benutzt das Unterprogramm GETLINE, um eine Textkette von der Tastatur in die Variable BUFFER einzulesen (beschrieben im Abschnitt "Ein Editor"). Abbruchbedingung für das Programm ist jetzt jedoch nicht ESCAPE, sondern die Taste CTRL-C. Diese Textkette wird gegen die Tabelle MNEMONIC verglichen, die einige Einträge mit Befehlen der Assemblersprache des 68000 enthält. Wenn die eingegebene Textkette einem Eintrag in der Tabelle entspricht, dann erfolgt bei FOUND zunächst die Textausgabe "Der Opcode ist $", gefolgt von der Anzeige eines entsprechenden Strings aus der Tabelle OPCODES. Das ist gewissermaßen die Übersetzung des Eingabestrings in

eine andere Sprache oder Aktion. – Kann zur Eingabe keine Entsprechung bei MNEMONIC gefunden werden, so erfolgt die Anzeige "Befehl unbekannt". In beiden Fällen geht es bei MAIN0 mit der Eingabe eines weiteren String weiter.

Bei der Untersuchung einer eingegebenen Textkette wird man normalerweise mit einem Filter auf das erste "erlaubte" Zeichen positionieren und wird auch die Frage stellen, was als Trenner hinter dem Eingabestring vorkommen darf, z. B. eine Null, ein Carriage Return, ein Zwischenraum, ein Komma oder ein Doppelpunkt (wie z. B. in BASIC, wenn mehrere Befehle in einer Zeile stehen). Es kommt also immer auf die Umgebung an. – Aus Gründen der Übersichtlichkeit wurde hier auf solche Feinheiten verzichtet: Für ein Schlüsselwort kann nur dann eine Entsprechung gefunden werden, wenn es in der ersten Schreibstelle des Buffers beginnt und wenn es nach hinten durch eine hexadezimale Null begrenzt ist.

Das Programm benutzt das Adreßregister A3 als Zeiger auf die Tabelle MNEMONIC mit den reservierten Befehlsworten, das Register A4 als Zeiger auf die Ausgabetexte in der Tabelle Opcodes und A5 als Zeiger in den Eingabepuffer. Diese Register werden ab Label MAIN2 entsprechend mit dem Befehl LEA (lade eine effektive Adresse) geladen.

Zum Lösungsweg und zu den Datenstrukturen folgende Überlegungen: Das Unterprogramm GETLINE begrenzt Eingaben mit einer hexadezimalen Null, die sich damit als Trenner anbietet, und zwar auch als Trenner zwischen den einzelnen Schlüsselworten in MNEMONIC. Diese Tabelle ist zur Markierung ihres Endes dort zudem mit zwei Nullen abgeschlossen, so daß wir beim Label PARSER feststellen können, ob der laufende Zeiger in A3 schon auf diese zweite Null weist.

Der entscheidende Vergleichsbefehl steht beim Label PARS2: Aus beiden Operanden (String im Buffer, Zeiger A5, und in MNEMONIC, Zeiger A3) wird ein Byte verglichen, und zwar in der Adressierung 'Adreßregister indirekt mit Postinkrement'.

D. h. nach dem Vergleich zeigen beide Register schon auf das nächste Byte im jeweiligen String. – Wenn die verglichenen Bytes ungleich sind, dann sind folgende Maßnahmen notwendig: Der nächste Vergleich mit einem Schlüsselwort muß wieder am Stringbeginn im Eingabepuffer beginnen, und zwar beim Label PARSER. Vorher muß jedoch auf das folgende Schlüsselwort in MNEMONIC weiterpositioniert werden. Auch muß Register A4 auf den nächsten Eintrag in OPCODE weisen. Und das geschieht in A3PLUS.

Am Label A3PLUS weist A3 bereits auf ein nachfolgendes Byte, das nicht mehr dem Vergleich beim Label PARS2 unterlag. Es können nun folgende Bedingungen außerdem gegeben sein: Das letzte Byte aus (A3) war eine Null. Dann zeigt A3 jetzt schon auf einen Buchstaben oder auf eine Null (Ende der Tabelle). Dann muß nicht weiterpositioniert werden, das Programm wird bei A3PLUS2 fortgesetzt. Die entsprechende Adressierung erfolgt mit –1(A3) (Adreßregister indirekt mit Offset oder Adreßdistanz –1). Es wird also das Byte getestet, das eine um 1 niedrigere Adresse hat, als jetzt in A3 enthalten.

Es mag aber auch sein, daß A3 derzeit auf eine Null als Trenner weist, dann muß A3 auf den nächsten Eintrag in MNEMONIC weiterpositioniert werden. Das geschieht beim Label A3PLUS0. Der dritte mögliche Fall bei Ungleichheit der Strings wird bei A3PLUS1 abgehandelt: Dort wird solange mit Postinkrement weiterpositioniert, bis das laufende Zeichen Null ist und A3 schon auf das 1. Byte des folgenden Tabelleneintrages weist.

Der vierte "Erfolgs-Fall" zeichnet sich beim Label PARS3 ab: Die Zeichen in den Strings sind bisher gleich gewesen. Nun kann der Vergleich abgebrochen werden, wenn beide Strings erschöpft sind. Die entsprechenden Feststellungen erfolgen bei den Labels PARS3, PARS4 und PARS5. In diesem Falle gelangt man zum Label FOUND mit Erfolgsmeldung, im anderen ist nach dem Anfangserfolg bei PARS2 weiterzusuchen, bis sich totale Gleichheit bis hin zu FOUND oder eine Abweichung ergibt.

Am Programm PARSER fällt auf, daß der eigentliche Parser mit den leistungsfähigen Adressierungsarten des 68000 mit wenigen Zeilen Quelltext auskommt. Weiterhin: Er läßt unterschiedliche Stringlängen sowohl im Eingabepuffer als auch in der Vergleichstabelle zu. Auf dieses einfache Beispiel lassen sich bequem verfeinerte Abfragestrukturen aufbauen. Und man sollte sich vor Augen halten, daß ein solcher Parser nicht nur mit einem Eingabestring von der Tastatur zusammenarbeitet, sondern mit beliebigen Strings, wenn man die Adreßregister entsprechend auf sie richtet.

PARSER

```
* PARSER
* Eingabestring mit Tabelle vergleichen
* Eingabe einer Zeile bis CR, Programmabbruch mit CTRL-C

 SECTION  ONE
main      movem.l   a3-a5,regsave  Register sichern
main0     move.l    #text1,d0      Zeiger auf interaktiven Text setzen
          jsr       printline      Ausgabe per GEMDOS
main1     jsr       crlf           Neue Zeile
          lea       buffer,a5      Benutze BUFFER fuer GETLINE
          clr.b     (a5)           Sicherheitshalber 00 dort einsetzen
* Das Unterprogramm GETLINE ist im Abschnitt "Ein Editor" beschrieben
          jsr       getline        1 Zeile in den Buffer lesen
          cmpi.b    #$03,d0        lag dort CTRL-C fuer Abbruch vor?
          beq.s     abbruch        ja, Ende des Programms
main2     lea.l     opcodes,a4     Zeiger auf den Beginn der Literale
          lea.l     mnemonic,a3    Zeiger auf Befehlstabelle
parser    tst.b     (a3)           Am Ende der Tabelle steht 00,00
          beq.s     notfound       Also nichts gefunden
          lea.l     buffer,a5      Vergleich vorbereiten, Pufferzeiger
pars1     tst.b     (a5)           Was steht im Puffer?
          beq.s     main0          nichts, neue Eingabe holen
pars2     cmp.b     (a3)+,(a5)+    Vergleiche Tabelle:Buffer
          bne.s     a3plus         ungleich, A3 hochzaehlen
pars3     tst.b     -1(a3)         gleich, war letztes Zeichen 00-Begrenzer?
          bne.s     pars5          nein, noch kein Ende bei MNEMONIC, aber??
pars4     tst.b     -1(a5)         Stringende auch im Buffer?
          bne.s     a3plus         nein, Eingabestring ist laenger
found     move.l    #text3,d0      beide Strings identisch inkl. Begrenzer
          jsr       printline      Anzeige: Schluesselwort gefunden
          move.l    a4,d0          Nun Opcode-String
          jsr       printline      ausgeben
          jsr       crlf           Neue Zeile
          bra.s     main0          Weitere Zeile holen

pars5     tst.b     -1(a5)         String im Buffer kann erschoepft sein
          bne.s     pars2          nein. Anderenfalls weiter in der Tabelle
a3plus    tst.b     -1(a3)         War es Ende eines Eintrags in der Tabelle?
          beq.s     a3plus2        ja
a3plus0   tst.b     (a3)+          Laufendes Zeichen 00?
          beq.s     a3plus2        ja
```

Fortsetzung

PARSER (Fortsetzung)

```
a3plus1   tst.b     (a3)+          Begrenzer suchen zwischen Mnemonics
          bne.s     a3plus1        Noch keine Null
a3plus2   add.l     #5,a4          Folgestring in den Opcodes adressieren
          bra.s     parser         Vergleich mit folgendem Eintrag in MNEMONIC
notfound  move.l    #text2,d0      Anzeige: Befehl unbekannt
          jsr       printline
          jsr       crlf
          bra.s     main0          Naechste Eingabe holen
abbruch   movem.l   regsave,a3-a5  Register zurueckholen
          jmp       term

 NOLIST
 INCLUDE  GEMDOSFN                 Einzubindendes Quellfile
 INCLUDE  FILTER
 LIST
* Textkonstanten
text1     dc.b      'Eingabe von Mnemonics',0
text2     dc.b      '  Befehl unbekannt',0
text3     dc.b      '  Der Opcode ist   $ ',0
* Nachfolgend einige mnemonische Befehle des 68000
mnemonic  dc.b      'nop',0,'reset',0,'rte',0
          dc.b      'rtr',0,'rts',0
opcodes   dc.b      '4e71',0,'4e70',0          Opcodes hier als Literal
          dc.b      '4e73',0,'4e77',0,'4e75',0  f. PRINTLINE
* Arbeitsspeicher
regsave   ds.l      3              Register-Ablage zur Sicherung
buffer    ds.b      100            Tastatur-Eingabe-Puffer
 END
```

Kapitel 35

Strings suchen

Vor allem in Textprogrammen und Datenbanken wird nach Zeichenketten (Strings) gesucht. Sonderformen sind dabei die Instring-Suche (ob eine Zeichenkette in einer anderen enthalten ist) und die Suche mit einem unvollständigen Suchbegriff, in dem Stellvertreter-Zeichen enthalten sind, an deren Stellen im gesuchten String beliebige Zeichen stehen dürfen. So findet man mit "Me@er" z. B. "Meier", "Meyer", "Meter" usw. – Die Abschnitte **SUCHLOOP** und **STELLV** im folgenden interaktiven Beispielprogramm beschreiben beide Algorithmen.

Überblick. Das Programm beginnt bei MAIN1 zunächst mit einer Texteingabe durch den Benutzer. Er kann viele Zeichen Text eingeben, der später auf Zeichenketten abgesucht wird. Bei der Einspeicherung werden Zeilen mit hex 00 begrenzt, weil sie sich dann später mit dem Unterprogramm PRINTLINE bequem ausgeben lassen. Man kann natürlich auch anders verfahren (siehe auch den Abschnitt "Ein Parser"). Die Eingabe wird abgeschlossen, indem man nur ein CR in eine neue Zeile tippt. Damit gelangt man zum Label BEFEHLE und kann mit 'n' die normale Instring-Suche beginnen (Label "SUCHLOOP"), mit 's' die Suche mit zugelassenen Stellvertreterzeichen (Label STELLV). Beide Abschnitte haben Ähnlichkeiten.

Programmablauf. Zunächst wird für beide Teile die Eingabe eines Suchstrings gefordert, der wie üblich mit GETLINE im Tastaturpuffer abgelegt wird. In beiden Teilen wird später der Ver-

gleich byte-weise von Speicher zu Speicher mit dem Befehl CMPM (A5)+,(A6)+ durchgeführt. A5 weist dabei auf den Tastaturpuffer und A6 in den Textspeicher mit den eingegebenen Zeilen. Die Länge des Suchstrings steht nach Ausführung von GETLINE im Register A4, dessen Inhalt als Schleifenzähler für den Befehl DBNE Dx,LOOP dient. Dieser Befehl verläßt die Suchschleife, wenn a) zwei Bytes nicht übereinstimmen oder b) wenn Dx negativ geworden ist. Das zieht folgende Programmierung nach sich: Dx ist nach der Übertragung der wirklichen Länge aus A4 um 1 zu vermindern. Wenn nun später die Suchschleife verlassen ist, so kennen wir noch nicht den Grund dafür. Wir müssen daher das Vorzeichen von Dx (bei uns mit TST.W D1) prüfen. Wenn es negativ ist, dann wissen wir, daß ein String im Textspeicher in ganzer Länge mit der Vorgabe übereinstimmte. Wir sind fündig geworden und können bei FOUND die gefundene Zeile ab Stringbeginn anzeigen und die Suche abbrechen.

Wenn das Register Dx positiv blieb, dann trat während des Stringvergleiches eine Ungleichheit ein. Wir sind jetzt bei NOTFOUND gezwungen, den Vergleich von einem neuen Startpunkt aus fortzusetzen. Dazu wird der in A2 bewahrte Startzeiger in den Text um 1 erhöht und darauf geprüft, daß er nicht das Ende der Textspeicherung in A3 überschreitet. Wenn nein, dann geht die Suche bei SUCHLOOP bzw. STELLV weiter. Dieses Hochsetzen des Zeigers in den Text um 1 führt im ungünstigsten Fall (der zu findende String steht am Textende) dazu, daß die Vergleichs-Schleife fast so oft durchlaufen werden muß, wie es der insgesamt vorhandene Zahl von Bytes im Text entspricht. Es gibt Lösungswege, um schneller zum Ziel zu kommen.

Die Suche mit Stellvertreter-Zeichen '@' hat folgende Besonderheit: Bei STELLV0 wird zunächst das aktuelle Zeichen im Suchpuffer geprüft: Ist es das Stellvertreter-Zeichen, dann gilt per Definition Gleichheit mit dem Zeichen im Textpuffer. Wir erhöhen daher die Zeiger auf beide Puffer in A5 und A6 und setzen mit MOVE.W #$04,CCR zwangsweise die Gleichheitsbedingung im Bedingungscode-Register, ehe wir unten in die Schleife bei STELLV1 weiterspringen.

FINDEN

```
* FINDEN
* Interaktive Texteingabe und Stringsuche

cr        equ       $0d            Carriage Return
lf        equ       $0a            Linefeed
escape    equ       $1b            Escape-Taste
space     equ       $20            Zwischenraum

 SECTION  ONE

main      movem.l   a3-a6,regsave  Register sichern
          lea.l     textbuf,a3     Zeiger auf Abspeicherungsstelle
main1     jsr       crlf           Neue Zeile
          move.l    #text1,d0      Ueberschrift anzeigen
          jsr       printline
          jsr       crlf
          lea.l     inbuff,a5      Zeige auf Tastaturpuffer
          move.b    #0,(a5)        Delimiter vorne in den Puffer
          clr.w     (a3)           0000 als Delimiter
getloop   lea.l     inbuff,a5      Zeige auf Tastaturbuffer
          clr.b     (a5)           Delimiter
          jsr       getline        Zeile holen
          cmp.b     #escape,d0     Will man abbrechen?
          beq       abbruch        ja
          jsr       crlf           Neue Zeile
          lea.l     inbuff,a5      Wieder auf Eingabepuffer zeigen
          tst.b     (a5)           Abbruch bei leerer Eingabe
          beq       befehle        nun zum Suchen
transp    move.b    (a5)+,(a3)+    Eingabestring in den Textbuffer
          bne.s     transp         Bis die Null auftaucht
          clr.b     (a3)           2. Null anhaengen
          bra.s     getloop        Weitere Zeilen holen

befehle   move.l    #text5,d0      Befehlsmenue anzeigen
          jsr       printline
          jsr       dircon         Befehlstaste holen
          cmp.b     #'w',d0        weitere Eingabe?
          beq       main1
          move.w    d0,key         Zeichen sichern
          move.l    #text2,d0      Suchstring anfordern
          jsr       printline
* Suchstring holen und  seine Parameter anlegen
          lea.l     inbuff,a5      Zeige auf Textbuffer
          jsr       getline        Suchstring holen
          cmp.b     #escape,d0     Will man abbrechen?
          beq       abbruch        ja
          jsr       crlf           Neue Zeile
          lea.l     inbuff,a5      Wieder auf Eingabepuffer zeigen
          tst.b     (a5)           Abbruch bei leerer Eingabe
          beq.s     befehle
* Rollenzuweisung der Register beim Suchen:
* A4 enthaelt Laenge des Suchstrings,
* A2 merkt die Einsatzstelle des Suchens in TEXTBUF
* A3 ist Zeiger auf Textende
* A6 ist Laufzeiger auf TEXTBUF
* D3 enthaelt das letzte Zeichen des Suchstrings
* D1 wird Schleifenzaehler fuer den Stringvergleich benutzt
```

Fortsetzung

FINDEN (Fortsetzung)

```
          move.b    1(a5,a4.w),d3  Lade letztes Zeichen Suchstring
          lea.l     textbuf,a2     Zeige auf Beginn des Textspeichers
          cmpi.b    #'n',key+1     Welcher Befehl?
          beq       suchloop       normales Suchen
          cmpi.b    #'s',key+1
          beq       stellv         Suche mit Stellvertreterzeichen '@'
          bne       befehle        hole besseren Befehl
* normales Suchen zu aufsteigenden Adressen
suchloop  move.l    a2,a6          Einsatzpunkt aktualisieren
          lea.l     inbuff,a5      Zeige auf Tastaturpuffer
          move.w    a4,d1          Stringlaenge
          sub.w     #1,d1          fuer Decrement and Branch
nuns2     cmpm.b    (a5)+,(a6)+    Vergleich zu aufsteigenden Adressen
          dbne      d1,nuns2
          tst.w     d1             String vollkommen erledigt?
          bpl.s     notfound
found     move.l    #text3,d0      Anzeige: gefunden
          jsr       printline
          move.l    a2,d0          Adresse Fundstelle
          jsr       printline
          jsr       crlf
          bra       befehle        anderen String suchen
notfound  cmpa.l    a2,a3          Ende Textspeicher?
          bcs.s     notf1          ja
          adda.l    #1,a2          Setze auf Folgebyte
          bra.s     suchloop       Suche fortsetzen
notf1     move.l    #text4,d0      Nichts gefunden
          jsr       printline      anzeigen
          adda.l    #1,a2          Einsatzpunkt weitersetzen
          jsr       crlf
          bra       befehle

* stellv = Suche mit Stellvertreterzeichen '@'
stellv    move.l    a2,a6          Einsatzpunkt aktualisieren
          lea.l     inbuff,a5      Zeige auf Tastaturpuffer
          move.w    a4,d1          Stringlaenge
          sub.w     #1,d1          fuer Decrement and Branch
stellv0   cmpi.b    #'@',(a5)      Stellvertreter im Suchstring?
          bne.s     stellv1        nein
          adda.l    #1,a5          Register weitersetzen
          adda.l    #1,a6          auf Folgezeichen
          move.w    #$04,ccr       Signalisiere Gleichheit des Vergleichs
          bra.s     stellv2        Ausgang, wenn '@' an letzter Stelle
stellv1   cmpm.b    (a5)+,(a6)+    Vergleich zu aufsteigenden Adressen
stellv2   dbne      d1,stellv0
          tst.w     d1             String vollkommen erledigt?
          bmi       found          String gefunden
          cmpa.l    a2,a3          Ende Textspeicher?
          bcs       notf1          ja, Anzeige, Abbruch
          adda.l    #1,a2          Setze auf Folgebyte
          bra.s     stellv         Suche fortsetzen

abbruch   movem.l   regsave,a3-a6
          jmp       term           Programm beenden

* Textkonstante
text1     dc.b      'Stringsuche, zunaechst Eingabe einiger Zeilen ',cr,lf
          dc.b      'Ende der Eingabe, wenn nur CR gedrueckt wird  ',cr,lf,0
text2     dc.b      'Suchstring eingeben:',cr,lf,0
text3     dc.b      'Gefunden wurde:   ',cr,lf,0
text4     dc.b      '       Nichts gefunden',cr,lf,0
text5     dc.b      'Befehle mit Tastendruck:',cr,lf
```

Fortsetzung

FINDEN (Fortsetzung)

```
          dc.b      ' w = weitere Texteingabe',cr,lf
          dc.b      ' n = normales Stringsuchen',cr,lf
          dc.b      ' s = Suchen mit Stellvertreter-Zeichen',cr,lf

* Arbeitsspeicher
regsave   ds.l      3             Ablage von Registern
filtflag  ds.w      1             Flag-Variable
key       ds.w      1             Befehlstaste merken
inbuff    ds.b      80            Allg. Eingabepuffer
textbuf   ds.b      10000         Textspeicher fuer 10000 Zeichen

 INCLUDE  GEMDOSFN
 END
```

SPEZIELLE EIN- UND AUSGABEN

SPEZIELLE EIN- UND AUSGABEN

Kapitel 36

Tastatur-Dekodierung

Das kleine Hilfsprogramm **HEXASCII** gibt zu jedem Tastendruck den im Datenregister D0 empfangenen hexadezimalen Wert als 2 ASCII-Bytes und außerdem den sog. Scancode auf die Konsole aus.

Überblick. Für die Abfrage der Tastatur benutzen wir die GEMDOS-Funktion **DIRCON**. Sie liefert in den Bits 0 ... 7 des Registers D0 den empfangenen hexadezimalen Wert des Zeichens ab und in den Bits 16 ... 23 den dazugehörigen Scancode. Beide Codes werden mit der allgemein verwertbaren Routine **HEXOUT** in 2 ASCII-Zeichen umgewandelt und auf den Bildschirm ausgegeben.

Das Studium der Zeichen-Dekodierung ist für die verschiedensten Aufgaben, z. B. für die Textverarbeitung interssant, weil immer wieder Zeichensätze für Drucker und andere Peripheriegeräte umgewandelt werden müssen.

Zum Programmaufbau. Das Programm benutzt die denkbar einfachsten Befehle: Mit dem Unterprogramm DIRCON (Zeile 7) werden die Codes geholt. In Zeile 8 wird mit dem Befehl CMPI (Vergleich mit Direktwert) geprüft, ob mit der Taste Control-C der Programmabbruch am Label ABBRUCH erwünscht ist. Zeile 10 bringt eine Kopie der empfangenen Codes auf den Stack. In 11 wird zunächst der Hexcode für das ASCII-Zeichen ausgegeben (BSR HEXOUT). Die Kopie der empfangenen Codes wird vom Stack zurückgeholt, und es werden die beiden Registerhälf-

ten von D0 mit SWAP vertauscht, weil im vorderen Teil der Scancode gestanden hat. Dieser gelangt nun ebenfalls mit BSR HEXOUT zur Ausgabe.

Das Unterprogramm HEXOUT findet sich in ähnlicher Form in allen Monitorprogrammen, und es wird uns noch öfter in Mitverwendungen wiederbegegnen. Ein Hexbyte wird in zwei Abschnitten zu je 4 Bits bearbeitet, die für die Werte 0 ... 9 und, mit einem Sprung von 7 in der ASCII-Tabelle, für die Werte A ... F in ein ASCII-Zeichen umgewandelt werden. Die eigentliche zeichenweise Ausgabe besorgt die GEMDOS-Funktion **CONOUT**.

Zum Programmablauf. Bei der Erprobung des Programms macht man folgende Feststellungen: Buchstaben- und Zahlentasten haben (wie erwartet), einen von Null verschiedenen ASCII- und Scancode. Der ASCII-Code entspricht dabei im allgemeinen dem Standard, jedoch mit Ausnahme der deutschen Sonderzeichen, für die wir folgende Verschlüsselung finden:

Zeichen	ä	ö	ü	ß	Ä	Ö	Ü
Hexcode	84	94	81	9E	8E	99	9A

Wenn man diese Sondertasten zusammen mit der Alternate-Taste betätigt, so findet man folgende normale ASCII-Dekodierung:

Zeichen	]	[	@	}	{	\
Hexcode	5D	5B	40	7D	7B	5C

Weitere Beobachtungen zu Sondertasten: Der mittlere Tastenblock mit Help, Undo, Insert und den Cursor-Tasten sowie die 10 Funktionstasten liefert in diesem Programm für den ASCII-Teil eine 00 ab. Die Unterscheidung dieser Tasten erfolgt nur durch den Scancode in den Bits 16 ... 23 des Registers D0. Daher nachfolgend eine Tabelle der Scancodes für alle Tasten, der u. a. zu entnehmen ist, daß der Scancode der 10 Funktionstasten F1 ... F10 unterschiedlich ist, wenn diese ohne oder zusammen mit der Shift-Taste betätigt werden, womit sich dann leicht 20 eigene Funktionen einrichten lassen.

HEXASC II

```
* Umwandlung Tastaturcode zur ASCII-Ausgabe
* Programmabbruch mit der Taste Control-C

 SECTION  ONE

space     equ       $20            Definition Zwischenraum

main      jsr       dircon         Zeichen holen per GEMDOS
          cmpi.b    #$03,d0        Ist es CTRL-C ?
          beq.s     abbruch        ja
          move.l    d0,-(sp)       Kopie des Zeichens sichern
          bsr.s     hexout         Ausgabe Konsole des ASCII-Teiles
          move.w    #space,d0      Space zwischenschieben
          jsr       conout         und Ausgabe
          move.l    (sp)+,d0       Kopie mit dem Scancode zurueckholen
          swap      d0             Registerhaelften vertauschen
          bsr.s     hexout         Scancode an Konsole ausgeben
          jsr       crlf           Vorschub in neue Zeile
          bra.s     main           weiter in der Schleife

abbruch   jmp       term           Programm per GEMDOS abbrechen

*         hexout gibt 1 Hexbyte als 2 ASCII-Zeichen aus
hexout    move.w    d0,-(sp)       Sichern einer Kopie von d0
          lsr.b     #4,d0          zuerst high byte isolieren
          bsr.s     hexout1        und ausgeben
          move.w    (sp)+,d0       Kopie holen
          andi.b    #$0f,d0        low Byte isolieren
hexout1   ori.w     #$30,d0        ASCII '0' hinzu-odern
          cmpi.b    #$3a,d0        groesser '9'?
          bcs.s     hexout2        nein
          addq.b     #7,d0         Korrektur fuer A-F
hexout2   jsr conout               Die eigentliche Ausgabe
          rts
 NOLIST
 INCLUDE  GEMDOSFN                 Siehe GEMDOS Toolkit
 END
```

Scan-Codes der Schreibmaschinen-Tastatur und der ungeshifteten Funktionstasten

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		ESC	1	2	3	4	5	6	7	8	9	0	ß		BS	TAB
1	Q	W	E	R	T	Z	U	I	O	P	Ü	+	RET		A	S
2	D	F	G	H	J	K	L	Ö	Ä	#		~	Y	X	C	F
3	B	N	M	,	.	–				SPACE		F1	F2	F3	F4	F5
4	F6	F7	F8	F9	F10			CLR	↑			←		→		
5	↓		INS	DEL												
6	<	UNDO	HELP													
7																

Scan-Codes des Zehnerblocks und der geshifteten Funktionstasten

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3												F1	F2	F3	F4	F5
4	F6	F7	F8	F9	F10						–				+	
5																
6				(	)	/	*	7	8	9	4	5	6	1	2	3
7	0	●	ENTER													

Scan-Codes Alternate und Schreibmaschinen-Tasten

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
7									1	2	3	4	5	6	7	8
8	9	0	ß													

Kapitel 37

Memory-Dump

Dieses etwas längere Programm bringt den hexadezimalen Inhalt von Speicherstellen zur Anzeige und zeigt außerdem deren Inhalt in der ASCII-Ausdeutung, soweit es sich um druckbare Zeichen handelt. Es kommen viele zusätzliche Befehle zum Einsatz: Vergleiche und davon abhängige Verzweigungen, logische Befehle sowie indirekte Adressierungen mit und ohne zusätzliches Indexregister. Daneben wird die XBIOS-Funktion **SUPEREXEC** zum Einsatz gebracht, die Programme im Supervisor-Modus auszuführen gestattet.

Das Hauptprogramm **MDUMP** gliedert sich in den Abschnitt MAIN mit den Interaktionen des Bedieners und den Abschnitt DUMP, der die Ausgabe per GEMDOS besorgt. Zu letzterem gehört das Unterprogramm DUMP3, das im Supervisor-Modus zum Ablauf gebracht wird. HEXOUT gibt wie üblich eine Hexzahl als 2 ASCII-Zeichen aus. Neu ist das Unterprogramm HEXFILT, das in der Variablen HEXFLAG angibt, ob eine Tastatureingabe als eine Hexziffer angesehen werden kann: Die Ziffern 0 ... 9 und die Buchstaben A ... F sowie a ... f.

Der Programmteil **MAIN** dürfte nach folgender Funktionsbeschreibung schnell durchschaubar sein: Das Variablenfeld Adresse wird mit CLR.L zu Null gesetzt, weil zu ihm später ab Label MAIN3 immer die Adreßeingaben 4-bitweise hinzugeodert werden. Zuvor muß natürlich der bisher aufgebaute Adressenteil um 4 Stellen nach links verschoben worden sein. Das geschieht mit dem Befehl ASL.L #4,D0.

Die Tastaturabfrage erfolgt ab Label MAIN2. Erlaubte Hexziffern führen nach der Filterung mit BSR HEXFILT weiter zu MAIN3. Es können bis zu acht Zeichen eingegeben werden. Ein Carriage Return darf die Eingabe vorzeitig beenden. Die Taste CTRL-C führt zum Programm-Abbruch. Mit ESCAPE kann eine neue Adressenvorgabe eingeholt werden. Die Plus- und die Minustasten führen zu einem Weiter- bzw. Rückwärtsblättern um jeweils 16 Adressen.

Eine Ausgabezeile wird ab Label DUMP1 erzeugt. Zunächst erfolgt die Anzeige der erreichten Adresse. Hier wird nun ganz umsichtig zunächst das Adreßregister A0 vorübergehend auf den Stack gerettet. A0 nimmt dann den Zeiger auf die Variable ADRESSE auf (Befehl LEA.L). Mit der indirekten Adressierung (A0) wären wir nun in der Lage, jeweils 1 Byte für die Ausgabe mit HEXOUT zu erfassen. Wir wollen aber in einer Zeile jeweils 16 Bytes zur Ausgabe bringen und müssen uns daher eine Automatik innerhalb einer Schleife überlegen. Dazu nehmen wir die Hilfsvariable ZIFFZAHL, die wir von 0 an hochzählen und jeweils am Schleifenende auf den Endwert 4 prüfen. Nun sind wir beim Label DUMP10 in der Lage, die Adressierung 'Adreßregister indirekt mit Index' zu wählen, nachdem wir das Indexregister D0 zuvor mit der Laufvariablen ZIFFZAHL geladen haben. Die effektive Adresse ergibt sich dort also aus der Basis ADRESSE + ZIFFZAHL.

Am Label DUMPla wird die Ausgabe von 16 Speicherzellen-Inhalten vorbereitet, und zwar mit der XBIOS-Funktion **SUPEREXEC**, die das Unterprogramm DUMP3 im Supervisor-Modus der CPU zur Ausführung bringt. Dazu folgende Hinweise: Es ginge auch ohne diese Konstruktion, wenn wir nicht auch in die geschützten Adressenbereiche von $0 ... $7FF hineinschauen wollten. Ohne SUPEREXEC würde unser Programm dann zwangsweise abgebrochen.

SUPEREXEC hat folgende Konstruktion: Man bringt die Adresse der auszuführenden Routine (die mit RTS abgeschlossen sein muß) auf den Stack, dann die Funktionsnummer 38 und ruft

schließlich TRAP #14 auf. nach der Ausführung ist der Stackpointer um 6 zu berichtigen.

Betrachten wir nun die Routine **DUMP3**: Wiederum ist das Adreßregister A0 vorübergehend zu sichern. Zwischenzeitlich wird es Zeiger auf den darzustellenden Adressenbereich, der in der Variablen ADRESSE vorgegeben ist. Am Label DUMP3A wählen wir nun wiederum eine indirekte Adressierung, um das darzustellende Byte zu erfassen, diesmal aber mit (A0)+. Der Inhalt von A0 wird also jeweils nach der ausführung um 1 erhöht (weil wir byte-weise arbeiten). Der in A0 erreichte Stand wird für die nächste Runde jeweils nach ADRESSE zurückgeschrieben. – Im weiteren Verlauf von DUMP3 wird in Abhängigkeit von ASCI-FLAG entschieden, ob wir im 1. oder 2. Aufruf der Routine sind. Im ersten werden die Hexbytes ausgegeben, im zweiten ihre Ausdeutung als ASCII-Bytes, soweit sie druckbar sind.

Die Routine **HEXFILT** ist ein Filter mit zahlreichen Vergleichen auf zulässige Wertebereiche, um festzustellen, ob ein Zeichen als hexadezimale Ziffer gedeutet werden kann. Filter dieser Art werden immer wieder bei Eingaben benutzt.

Im Programmverlauf haben wir Unterprogrammaufrufe sowohl mit BSR wie auch mit JSR bewirkt. Dabei konnten wir BSR benutzen, weil die Einsprungsstellen nie mehr als 32KB auseinanderliegen. JSR wäre hier also nicht nötig gewesen, was jedoch in großen Programmen durchaus der Fall sein kann. Der Assembler gibt dann einen Fehlerhinweis, wenn zu weite Verzweigungen beabsichtigt sind.

Zur Hantierung des Programms ist zu bemerken, daß ein Absturz des Computers eintreten kann, wenn die Tasten +, – und CR für die Wiederholung der Ausgaben zu schnell aufeinanderfolgend betätigt werden. Die Ursachen sind nicht in der Logik des Programmes, sondern im Betriebssystem zu suchen, denn bei einer Verlangsamung des Tastatur-Repeats (im Kontrollfeld) treten keine Überraschungen ein.

MDUMP

```
* MDUMP, hexadezimale Speicheranzeige
* Programmabbruch mit der Taste Control-C
* Weiterblaettern mit der +-Taste

 SECTION  ONE
space     equ       $20            Definition Zwischenraum
escape    equ       $1b            und Escape-Taste

main      clr.l     adresse        Adresse = 00
          clr.w     ziffzahl       Start mit 0 Ziffern
main1     bsr       crlf           Neue Zeile
          move.l    #text,d0       Zeiger auf interaktiven Text setzen
          jsr       printline      Ausgabe per GEMDOS
main2     jsr       dircon         Zeichen holen per GEMDOS
          cmpi.b    #$03,d0        Ist es CTRL-C ?
          beq       abbruch        ja
          cmpi.b    #escape,d0     Bei Escape neue Adresse
          beq.s     main           anfordern
          cmpi.b    #$0d,d0        Carriage Return?
          beq       dump           Dann immer ausgeben
          cmpi.b    #'+',d0        Weiterblaettern?, Taste '+'
          beq       dump           ja
          cmpi.b    #'-',d0        '-' = zurueckblaettern
          beq       zurueck        ja
          bsr       hexfilt        Hex-Zeichen?
          tst.b     hexflag
          bmi.s     main2          besseres Zeichen holen
          move.w    d0,-(sp)       Zeichen sichern
          cmpi.b    #':',d0        Buchstaben zu Hex korrigieren
          bcs.s     main3          Ziffer empfangen
          addi.b    #9,d0          Buchstaben korrigiert
main3     andi.w    #$f,d0         Zeichen maskieren
          move.b    d0,savchr      Kopie des Zeichens merken
          move.w    (sp)+,d0
          bsr       conout         Zeichenausgabe, alter Wert noch in D0
          move.l    adresse,d0     Erreichte Adresse holen
          asl.l     #4,d0          Empfangsfeld vorbereiten
          or.b      savchr,d0      4 Bits ergaenzend hinzu-odern
          move.l    d0,adresse     Erreichte Adresse merken
          addq.w    #1,ziffzahl    Maximal 8 Ziffern, von 0...7
          cmpi.w    #8,ziffzahl
          bne.s     main2          Weitere Ziffern holen

dump      jsr       crlf           Vorschub in neue Zeile
          clr.w     ziffzahl       Hilfszaehler=0, nun Adresse ausgeben
          clr.b     asciflag       Zuerst Hexdump
dump1     move.l    a0,-(sp)       Adressregister A0 sichern
          lea.l     adresse,a0     Zeiger auf Adressfeld
          move.w    ziffzahl,d0    D0 wird Indexregister
dump10    move.b    (a0,d0.w),d0   Inhalt holen, indirekt A0 mit Index D0
          move.l    (sp)+,a0       Adressregister zurueck
          bsr       hexout         Ausgabe als 2 ASCII-Zeichen
          addq.w    #1,ziffzahl    ziffzahl=ziffzahl+1
          cmpi.w    #4,ziffzahl    Nur Bytes 0...3 nehmen
          bne.s     dump1          Adresse weiter ausgeben
          move.b    #space,d0      Zwischenraum
          jsr       conout         Space hinter Adresse
dump1a    move.w    #16,ziffzahl   Hilfszaehler einrichten
```

Fortsetzung

MDUMP (Fortsetzung)

```
dump2     move.l    #dump3,-(sp)    SUPEREXEC im XBIOS vorbereiten, Adresse
          move.w    #38,-(sp)       Funktionsnummer
          trap      #14             XBIOS-Aufruf, Routine dump3 ausfuehren
          addq.l    #6,sp           Stack berichtigen
          subq.w    #1,ziffzahl
          bne       dump2           Weitere Anzeigen
          tst.b     asciflag        Schleife verlassen?
          bmi.s     dump2a          Ja
          not.b     asciflag        Negativ machen
          subi.l    #16,adresse     Wieder an den Zeilenstart gehen
          bra.s     dump1a          Jetzt ASCII-Ausgabe
dump2a    jmp       main2

zurueck   subi.l    #32,adresse     Adresse-32 setzen, Taste '-'
          bra       dump

abbruch   jmp       term            Programm per GEMDOS abbrechen

dump3     move.l    a0,sava0        Register sichern
          move.l    adresse,a0      Adresse laden
dump3a    move.b    (a0)+,d0        Inhalt unter Adresse holen
          move.l    a0,adresse      Erhoehte Adresse merken
          move.l    sava0,a0        A0 wieder herrichten
          tst.b     asciflag        Hex oder ASCII ausgeben?
          bpl.s     dump4
          cmpi.b    #$20,d0         Zeichen druckbar?
          bcc.s     dump6           Ja
          move.b    #'_',d0         Nein, Underline ausgeben
          bra.s     dump6
dump4     jsr       hexout          Ausgabe des Inhaltes
dump5     move.b    #space,d0       Space dahinter
dump6     jsr       conout          Ausgabe per GEMDOS
          rts

*         hexout gibt 1 Hexbyte als 2 ASCII-Zeichen aus
hexout    move.w    d0,-(sp)        Sichern einer Kopie von d0
          lsr.b     #4,d0           zuerst high byte isolieren
          bsr.s     hexout1         und ausgeben
          move.w    (sp)+,d0        Kopie holen
          andi.b    #$0f,d0         low Byte isolieren
hexout1   ori.w     #$30,d0         ASCII '0' hinzu-odern
          cmpi.b    #$3a,d0         groesser '9'?
          bcs.s     hexout2         nein
          addq.b    #7,d0           Korrektur fuer A-F
hexout2   jsr conout                Die eigentliche Ausgabe
          rts

* hexfilt zeigt in hexflag an, ob ein Zeichen in Gross-
* oder Kleinschreibung eine Hexziffer ist

hexfilt   clr.b     hexflag         Setze definiert zu 00
          cmpi.b    #'g',d0         Kleines g und folgende Zeichen
          bcc.s     nothex          sind nicht hex
          cmpi.b    #'a',d0         a ... f
          bcc.s     ishex           sind zugelassen
          cmpi.b    #'G',d0         Grosses G ...
          bcc.s     nothex          sind nicht hex
          cmpi.b    #'A',d0         A ... F, Grossbuchstaben
```

Fortsetzung

MDUMP (Fortsetzung)

```
        bcc.s     ishex        wieder zugelassen
        cmpi.b    #':',d0      Dazwischen liegen wieder keine
        bcc.s     nothex       Hexziffern
        cmpi.b    #'0',d0      Ziffernbereich 0 ... 9 erlaubt
        bcs.s     nothex       Zeichen liegt darunter
ishex   rts                    hexflag bleibt 00
nothex  not.b     hexflag      Flag wird $FF
        rts

text    dc.b      'Memory Dump ab Adresse ?   ',0

* Arbeitsspeicher
adresse ds.l      1            1 Langwort reservieren
sava0   ds.l      1            Adressregister sichern
ziffzahl ds.w     1            Zahl der Eingabeziffern
hexflag ds.b      1            Flag-Variable
asciflag ds.b     1            Steuerung der Ausgabe
savchr  ds.b      1            Fuer empfangenes Zeichen

 NOLIST
 INCLUDE GEMDOSFN               Siehe GEMDOS Toolkit
 END
```

Kapitel 38

Floppy-Test

Im GEMDOS sind zahlreiche Funktionen für den Umgang mit der Massenspeicherung auf Floppy Disk enthalten. In einem ersten einfachen Programm wird gezeigt, wie man ein File auf Floppy öffnet, liest und wieder schließt.

Im GEMDOS finden wir etwa 25 Funktionen für die Auswahl eines Laufwerkes, die Anlage eines neuen Files (CREATE), für das Öffnen und Schließen (OPEN und CLOSE) von Dateien, für das Lesen und Schreiben (READ und WRITE), für die Ansprache und Änderung des Inhaltsverzeichnisses und z. B. für die Umlenkung des E/A-Stromes. Auch diese Funktionen werden wie die etwas einfacheren (siehe Abschnitt "Ein GEMDOS-Toolkit") mit Parameterübergabe auf dem Stack und TRAP #1 aufgerufen. Die Funktionen sind so leistungsfähig, daß wir uns als Programmierer im allgemeinen nicht um Spuren und Sektoren auf der Diskette kümmern müssen. Damit ist bereits der Weg zur Massenspeicherung vorbereitet, wie sie von den meisten Anwenderprogrammen benötigt wird.

Im nachstehenden Programm werden die Grundfunktionen der Dateieröffnung (JSR FOPEN), des Lesens (JSR FREAD) und des Schließens (JSR FCLOSE) in den wenigen Zeilen zwischen den Labeln MAINOPEN und MAINCLOSE zur Ausführung gebracht. Die hierbei benutzten drei Unterprogramme folgen zwar einer noch zu erklärenden Syntax, sind aber in sich selbst wieder denkbar einfach. – In der Schleife MAINLOOP wird der in BUFFER eingelesene Text von 256 Zeichen mit JSR CONOUT zur Anzeige gebracht. Das nachfolgende JSR DIRCON wartet eine

Tastaturbetätigung ab, um dem Benutzer Zeit zur Betrachtung der Ausgabe zu geben.

Wenn man eine von CTRL-C verschiedene Taste betätigt, dann werden weitere 256 Bytes von der Floppy gelesen. Es erfolgen allerdings keine Fehleranzeige und auch kein automatischer Stop, wenn die einzulesende Datei bereits erschöpft ist. Das hierin liegende kleine Problem kann (in späteren Beispielen) dadurch beseitigt werden, daß man die Dateilänge aus ihrer Kopfinformation berücksichtigt. – Auch hinsichtlich des Dateinamens haben wir uns eine später zu beseitigende Beschränkung auferlegt. Man wird in vielen Fällen wie hier natürlich mit einer Textkonstanten arbeiten wollen. In anderen Fällen muß FILENAME jedoch eine Variable sein, die in einer interaktiven Tastaturabfrage beschickt wird. Eine weiter Beschränkung liegt in der Größe 256 für den BUFFER. Man arbeitet schneller und mit weniger Floppyzugriffen, wenn man den Puffer größer anlegt. Aber es geht ja hier um einen ersten Versuch, an dem der Leser die einfachen Prinzipien nachvollziehen kann.

Ein wichtiger neuer Begriff im Zusammenhang mit den Floppy-Funktionen des GEMDOS ist der "handle" (engl.: Handgriff oder Klinke). In den Kommentaren ist er hier als "Kanal-Nummer" bezeichnet worden, was den Verhältnissen wohl besser gerecht wird: Wenn man mit FOPEN eine Datei eröffnet, dann erhält man im Register D0.W eine Nummer zugeteilt, den handle, mit der man diese Datei künftig vereinfacht ansprechen kann, so wie es dann bei FREAD und FCLOSE geschieht. Diese Nummer entspricht einem logischen Kanal, und man tut gut daran, sie in einer Variablen abzuspeichern. – Ein weiterer Begriff ist der sog. "pathname" (Zugangspfad) einer Datei, zu dem ihre Laufwerksbezeichnung, der Name von "Ordnern" (und ggfs. Unter-Ordnern) sowie der Dateiname selbst gehören. Die Syntax dazu wird an anderer Stelle dargestellt.

Zu erklären bleiben jetzt die GEMDOS-Funktionen **FOPEN**, **FREAD** und **FCLOSE** mit den hexadezimalen Aufruf-Nummern $3D, $3F und $3E.

FOPEN verlangt zunächst ein Datei-Attribut auf dem Stack, das wie folgt definiert ist: 0 = nur Lesedatei, 1 = nur Schreibdatei und 2 = Schreib-/Lesedatei. Zu folgen hat der Adreßzeiger auf den String des Dateinamens (Filename) als Langwort. Nach dem GEMDOS-Aufruf erhält man den "handle" zurück oder einen negativen Wert, wenn ein Fehler eintrat (z. B. Datei nicht vorhanden).

FREAD verlangt einen Adreßzeiger auf den Puffer, in den gelesen werden soll, dann die Anzahl der zu lesenden Bytes und schließlich die Kanalnummer (handle). Wenn D0 nach der GEMDOS-Ausführung negativ ist, dann ist ein Lesefehler eingetreten, daher der abschließende Befehl TST.W D0 in diesen Routinen.

FCLOSE ist im Aufbau noch einfacher: Neben dem handle ist nur die Funktionsnummer zu übergeben. Auch hier wird im letzten Befehl das Vorzeichen des Registers D0 geprüft, um einen möglichen Fehlerhinweis zu erhalten. Der mit dem Befehl TST gewonnene Status im Bedingungscode-Register (CCR) der CPU wird durch den Folgebefehl RTS nicht verändert, der Status wird also getreu an die aufrufende Ebene zurückgemeldet.

Die genannten drei GEMDOS-Routinen lassen sich ebenso wie frühere in einer etwas verallgemeinerten Form in ein Toolkit für Floppyfunktionen einbringen.

FLOPPY 1

```
* FLOPPY1
* Versuche, ein File zu oeffnen und zu schliessen
* Programmabbruch mit CTRL-C

 SECTION  ONE
main      bsr       crlf          Neue Zeile
          move.l    #text1,d0     Zeiger auf interaktiven Text setzen
          jsr       printline     Ausgabe per GEMDOS
          jsr       crlf          Neue Zeile
mainopen  jsr       fopen         Oeffne vorhandene Datei
          bmi       openerr       Fehlerbehandlung
mainread  jsr       fread         Hole 256 Bytes in den Buffer
          bmi       readerr       Fehlerbehandlung
          lea.l     buffer,a5     Buffer-Adresse laden nach a5
          clr.w     d6            d6 als Aufwaertszaehler
```

Fortsetzung

FLOPPY 1 (Fortsetzung)

```
mainloop  move.b   (a5)+,d0        Hole 1 Byte aus dem Buffer
          jsr      conout          Ausgabe Bildschirm
          addq.w   #1,d6           Aufwaertszaehler+1
          cmpi.w   #256,d6         Bufferende erreicht?
          bcs      mainloop        noch nicht
main2     jsr      dircon          Anhalten und Zeichen holen per GEMDOS
          cmpi.b   #$03,d0         Ist es CTRL-C ?
          bne      mainread        Nein, Weiterlesen!
mainclose jsr      fclose          Datei schliessen
          bmi      closerr         Fehlerbehandlung
abbruch   jmp      term            Abbruch per GEMDOS
openerr   move.l   #ermesopn,d0    Zeiger auf Text setzen
openerl   jsr      printline       Ausgabe per GEMDOS
          bra      abbruch
readerr   move.l   #ermesrd,d0     Dito bei Lesefehler
          jsr      printline
          bra      mainclose       Aber noch CLOSE File
closerr   move.l   #ermescls,d0    Fehler bei CLOSE anzeigen
          bra      openerl

fopen     move.w   #2,-(sp)        Datei-Attribut R/W
          move.l   #filename,-(sp) Zeiger auf Dateinamen
          move.w   #$3d,-(sp)      Funktions-# FOPEN
          trap     #GEMDOS         Aufruf
          addq.l   #8,sp           Stack berichtigen
          move.w   d0,handle       Kanal-# merken
          tst.w    d0              Fehler, wenn negativ
          rts                      Zurueck mit Status

fclose    move.w   handle,-(sp)    Kanal nennen
          move.w   #$3e,-(sp)      Funktionsnummer
          trap     #GEMDOS         Aufruf
          addq.l   #4,sp           Stack berichtigen
          tst.w    d0              Fehlerstatus, wenn <00
          rts

fread     move.l   #buffer,-(sp)   Adresse Eingabe-Buffer f. Floppy
          move.l   #$100,-(sp)     Lies 256 Bytes zur Probe
          move.w   handle,-(sp)    Kanal-Nr.
          move.w   #$3f,-(sp)      Funktions-# FREAD
          trap     #GEMDOS         Aufruf
          add.l    #12,sp          Stack berichtigen
          tst.w    d0              Fehlerstatus erfassen
          rts

textl     dc.b     'Laden und Anzeigen eines Textfiles',0
filename  dc.b     'Floppy1.ASM',0
ermesopn  dc.b     'Cannot open File',0
ermescls  dc.b     'Cannot close file',0
ermesrd   dc.b     'Read Error',0

* Arbeitsspeicher
handle    ds.w     1               # des Kanals
buffer    ds.b     $100            ;fuer den Versuch
 NOLIST
 INCLUDE  GEMDOSFN                 Siehe GEMDOS Toolkit
 END
```

Kapitel 39

Floppy-Funktionen

GEMDOS enthält zahlreiche Funktionen für die Ansprache einer Floppy Disk. Der laufende Abschnitt erweitert unser Toolkit um das Programmpaket **GEMDOSF**, in dem wiederum zahlreiche Unterprogramme mitgeteilt werden, die man einer Programmbibliothek einverleiben sollte. Nicht enthalten sind Funktionen, die man normalerweise vom Desktop (interaktives Schreibtischprogramm des GEM) aus ausführt, wie z. B. das Formatieren, Umbenennen oder Kopieren.

Der Abschnitt "Floppy-Test" zeigte bereits, wie einfach eine mit Namen bestimmte Datei gelesen und wie ihr Inhalt zur Anzeige gebracht werden kann. Wir sollten uns nun Werkzeuge verschaffen, mit denen wir alle wichtigen Arbeiten mit einer Floppy anfassen können, insbesondere das Schaffen einer neuen Datei mit Namen und das Schreiben auf diese. Auch sollten wir mit Dateinamen beweglich umgehen können. Deswegen wurde im Abschnitt "Ein Editor" das Unterprogramm GETLINE dargestellt, mit dem beliebige Textketten von der Tastatur eingeholt werden können, die man natürlich auch für die Ansprache von Dateinamen benutzen kann.

Alle Routinen rufen das GEMDOS mit TRAP #1 auf und werden als Unterprogramm mit JSR in Aktion gerufen. Nun Einzelheiten mit Hinweis z. T. auf die Kommentare in den Unterprogrammen: Floppylaufwerke erhalten einen Buchstaben als Bezeichnung. Auf dem Desktop sehen wir die Laufwerke 'A' und 'B'

als Ikonen dargestellt. 'A' hat dabei die binäre Nummer 1 in einer binären Strichliste, die sich DRIVEMAP nennt, 'B' die Nummer2, bis hin zu 'P' mit Nr. 32, was 32 Bits entspricht oder einem Langwort. Wenn ein Laufwerk aktiviert worden ist, dann ist sein Bit zu '1' gesetzt. Normalerweise, wenn die Laufwerke 'A' und 'B' im Desktop aktiviert sind, findet man also eine '3' in der DRIVEMAP. Die Routinen SETDRIV und GETDRIV dienen nun dazu, ein Laufwerk zu aktivieren oder die aktuelle Aktivierung abzufragen. Vor der Ansprache einer Einheit sind wir also z. B. mit den Befehlen zur Bitprüfung in der Lage, die Ansprachemöglichkeit zu checken. Diese Routinen sind daher als Vorprogramme für höhere Dienstleistungen im GEMDOS-Paket zu betrachten.

Die gleiche Unterordnung gilt für die Funktionen SETDTA und GETDTA. Der Kürzel "DTA" steht dabei für 'Disk Transfer Address', einen Datenbereich von 44 Bytes, auf den ein Zeiger für das Setzen oder Lesen von Datei-Parametern gesetzt werden kann. Im Arbeitsspeicher haben wir diesen Bereich DTABUFF genannt. Wenn wir uns seine Parameter in der Programmliste ansehen, so finden wir ab DTABUF zunächst 21 unbenutzte Bytes, dann Datei-Attribute (Lesen/ Schreiben usw.), Erstellungsdaten, Dateigröße und Dateinamen. Die hier enthaltene Information entspricht damit der Anzeige des Desktop-Menues unter "Datei, Info-Anzeigen". Die Bedeutung der Attribute: $00 = Zugriff zum Lesen/Schreiben, $01 = schreibgeschützt, $10 = Ordner.

Die Funktion DFREE beschafft Information über den freien Speicherplatz einer Diskette und transportiert sie in den DFREEBUF unseres Arbeitsspeichers. Sie wird mit einer Laufwerksnummer (0 = aktuelles Laufwerk, 1 = A, 2 = B usw.) und dem Zeiger auf den DFREEBUF aufgerufen und liefert dort 4 Langworte ab. Das erste enthält die Zahl der freien, das zweite die Gesamtzahl der "Allocation Units". Jeweils zwei Diskettensektoren bilden eine solche "Einheit". – Das dritte Langwort enthält die Zahl der Bytes/Sektor und das vierte schließlich, wieviele (2) Sektoren eine Allocation Unit bilden. Aus diesen Parametern kann man die Zahl der noch freien Bytes aus den Variablen berechnen:

FREALUN*PHYSSEC* BYTSPSEC. – Mit der Funktion DFREE prüft man vor einer Abspeicherung, ob der Platz auf der Diskette noch ausreicht.

Das erstmalige Anlegen einer Datei erfolgt mit der Funktion $3C, CREATE. Diese Grundfunktion haben wir in der Programmliste zur Bequemlichkeit des Benutzers gleich in zwei Ausführungen codiert: **CREATERW** dient der Anlage einer Schreib-/Lesedatei und **CREATERD** legt eine Nur-Lesedatei an. Es wird jeweils das entsprechende Attribut auf den Stack gebracht, danach der Zeiger auf den zu verwendenden Dateinamen. Man erhält eine logische Dateinummer als "handle" zurück, die später beim Lesen oder Schreiben in den Funktionen FREAD, FWRITE und schließlich in FCLOSE zu benutzen ist.

Im Desktop-Menue findet der Leser die Möglichkeit, einen neuen Ordner anzulegen, der zunächst mit nichts gefüllt ist. Diese Dienstleistung erhalten wir durch die GEMDOS-Funktion MKDIR. Sie verlangt den Zeiger auf einen Dateinamen, der hier in FILENAME zu übergeben ist.

Auch die Funktion FOPEN zum Öffnen eines schon bestehenden Files wurde gleich in drei Versionen codiert: **FOPENWRIT** dient dem Schreiben, **FOPENRW** dem Lesen und Schreiben und **FOPENREAD** dem Lesen. Auch hier werden das Attribut und der Zeiger auf den Dateinamen auf dem Stack übergeben. Es ist darauf hinzuweisen, daß man auch hier eine logische Dateinummer vom Betriebssystem zurückerhält. – **FCLOSE** zum Schließen einer Datei benutzt als Parameter nur den aktuellen "handle".

FREAD zum Lesen aus einer Datei wurde bereits im Abschnitt "Floppy-Test" besprochen. **FWRITE** zum Schreiben läuft analog: Zu übergeben sind der Zeiger auf den Ausgabe-Puffer, die Menge der zu schreibenden Bytes und die logische Dateinummer, die wir auch Kanal genannt haben.

Innerhalb einer Datei kann mit der Grundfunktion FSEEK positioniert werden, die wir ebenfalls in drei Ausführungen abdruk-

ken: **FSEEK** zum Weitersetzen ab laufender Position um eine Menge Bytes (diese Menge ist im Register D0.L vor dem Aufruf als "Offset" zu übergeben), **FSEKKBEG** zum Positionieren mit Zählung ab Dateibeginn und **FSEEKEND** ab Dateiende.

Katalogfunktionen werden mit den Unterprogrammen **SFIRST, SNEXT, DSETPATH** und **DGETPATH** ausgeführt. SFIRST dient dem erstmaligen Suchen einer Datei mit Attribut (Lesen/Schreiben usw.) und der Übergabe eines Zeigers (auf DTABUFF), wo die erhaltene Information abgespeichert werden soll. Diese Dienstleistung lassen wir von der Funktion SETDATA (s. o.) ausführen. Schließlich wird der Zeiger auf einen Dateinamen verlangt, der vor dem Funktionsaufruf in der Variablen FILENAME enthalten sein muß. In diesem Suchnamen dürfen Stellvertreterzeichen (?) und Sterne (*) zur Abkürzung in der üblichen Form enthalten sein, z. B. *.ASM (Dateien mit der Erweiterung ASM im Namen), ebenso Pfadnamen, die über Ordner hinwegführen. – Im Register D0.W erhält man Nachricht, ob die Suche erfolgreich war. Wenn ja, dann ist der DTABUFF entsprechend mit Parametern gefüllt und man kann prüfen, ob man das so gefundene File ggfs. öffnen will. – SNEXT wiederholt die mit SFIRST begonnene Suche mit alten Parametern und liefert ggfs. in DTABUFF entsprechend ab.

DSETPATH macht ein untergeordnetes Inhaltsverzeichnis, das erst nach dem Öffnen von Ordnern aufgeschlagen werden kann, zu aktuellen Inhaltsverzeichnis. Die Vorgabe des Pfadnamens erfolgt in der Variablen FILENAME. DGETPATH ist die Umkehrung dazu: Man erhält in FILENAME den Pfadnamen, der zu aktuellen Inhaltsverzeichnis geführt hat.

Die Funktionen GETNSTDH und SETNSTH dienen der Umlenkung der Ein- und Ausgabe. Sie und die anderen vorgenannten GEMDOS-Funktionen werden in späteren Abschnitten dieses Buches zur Anwendung gebracht.

GEMDOSF

```
* GEMDOSF
* GEMDOS-Funktionen fuer Floppy

GEMDOS    equ       1

 SECTION  ONE

* Drive-# muss als Hex-Zeichen in D0.W sein
setdrv    move.w    d0,-(sp)        Parameteruebergabe
          move.w    #$0e,-(sp)      Funktionsnummer 14
          trap      #GEMDOS         Aufruf
          addq.l    #4,sp           Stack berichtigen
          move.l    d0,drivemap     Bitmap der Drives kommt zurueck
          rts

getdrv    move.w    #$19,-(sp)      Hole aktuellen Drive, Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #2,sp           Stack berichtigen
          move.w    d0,currdrv      # Current Drive merken
          rts

* Definiere den DTA-Buffer fuer File-Parameter
setdta    move.l    #dtabuff,-(sp) Adresse 44 Byte DTA-Buffer uebergeben
setdta1   move.w    #$1a,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #6,sp           Stack berichtigen
          rts

* Hole Adresse des aktuellen DTA-Buffers
getdta    move.w    #$2f,-(sp)      Funktionsnummer
          trap      #GEMDOS         Aktuelle DTA holen
          addq.l    #2,sp           Stack berichtigen
          move.l    d0,currdta      Adresse aktueller DTA merken
          rts

* Hole Drive-Parameter, Berechnung der noch freien Bytes
dfree     move.w    #0,-(sp)        Freien Speicher Current Drive bestimmen
          move.l    #dfreebuf,-(sp) Adresse Information Return Buffer
          move.w    #$36,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #8,sp           Stack berichtigen
          rts                       Information jetzt in DFREEBUF

* Lege neues File mit Namen und Attribut an
* Loesche ggfs. vorhandenes File gleichen Namens
createrw  move.w    #0,-(sp)        Setze Attribut Lesen/Schreiben
          bra.s     create1         Verzweige immer, kurz
createrd  move.w    #1,-(sp)        Setze File-Attribut nur Lesen
create1   move.l    #filename,-(sp) Zeiger auf zu verwendenden Dateinamen
          move.w    #$3c,-(sp)      Funktionsnummer
          trap      #GEMDOS         Aufruf
          addq.l    #8,sp           Stack berichtigen
          move.w    d0,handle       Kanal-# merken
          rts                       Fehler, wenn negativ

* Lege einen neuen Ordner/Sub-Directory an (Pfadnamen schaffen)
mkdir     move.l    #filename,-(sp) Zeiger auf Namen des Ordners
          move.w    #$39,-(sp)      Funktions-Nr.
          trap      #GEMDOS         Aufruf
          addq.l    #6,sp           Stach berichtigen
          tst.w     d0              Fehler, wenn negativ
          rts
```

Fortsetzung

GEMDOSF (Fortsetzung)

```
* SETATTRIB setzt ein File File-Attribut,
* das beim Aufruf in D0.W enthalten sein muss
setattrib move.w    d0,-(sp)        Attribut aus D0
          move.w    #1,-(sp)        Funktion: Setzen
setattrl  move.l    #filename,-(sp) Zeiger auf den Pfadnamen
          move.w    #$43,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          add.l     #10,sp          Stack berichtigen
          tst.w     d0              Fehler, wenn negativ
          rts

* GETATTRIB holt das Attribut einer Datei mit Pfadnamen
getattrib move.w    #0,-(sp)        Parameter unbedeutend
          move.w    #0,-(sp)        Funktion: Holen
          bra.s     setattrl        Attribut am Ende in D0 oder negativ

* Oeffne vorhandene Datei mit Namen und Attribut
fopenwrit move.w    #1,-(sp)        Oeffne zum Schreiben einer Datei
          bra.s     fopenl
fopenrw   move.w    #2,-(sp)        Oeffne zum Lesen und Schreiben
          bra.s     fopenl
fopenread move.w    #0,-(sp)        Oeffne zum Lesen
fopenl    move.l    #filename,-(sp) Zeiger auf Dateinamen
          move.w    #$3d,-(sp)      Funktions-# FOPEN
          trap      #GEMDOS         Aufruf
          addq.l    #8,sp           Stack berichtigen
          move.w    d0,handle       Kanal-# merken
          tst.w     d0              Fehler, wenn negativ
          rts                       Zurueck mit Status

* Schliesse Datei mit handle-#
fclose    move.w    handle,-(sp)    Kanal nennen
          move.w    #$3e,-(sp)      Funktionsnummer
          trap      #GEMDOS         Aufruf
          addq.l    #4,sp           Stack berichtigen
          tst.w     d0              Fehlerstatus, wenn negativ
          rts

* Einlesen von einer Datei mit handle-# ab Adresse bufptri
* Die zu lesende Menge muss in BUFLENI uebergeben sein
fread     move.l    bufptri,-(sp)   Adresse Eingabe-Buffer f. Floppy
          move.l    bufleni,-(sp)   Vorgabe der Menge Bytes
          move.w    handle,-(sp)    Kanal-Nr.
          move.w    #$3f,-(sp)      Funktions-# FREAD
          trap      #GEMDOS         Aufruf
          add.l     #12,sp          Stack berichtigen
          tst.w     d0              Fehlerstatus erfassen
          rts

* Schreiben auf eine Datei mit handle-# ab Adresse bufptro
                              s in BUFLENO uebergeben sein

* Schreiben auf eine Datei mit handle-# ab Adresse bufptro
* Die zu schreibende Menge muss in BUFLENO uebergeben sein
fwrite    move.l    bufptro,-(sp)   Adresse Ausgabe-Buffer f. Floppy
          move.l    bufleno,-(sp)   Vorgabe der Menge Bytes
          move.w    handle,-(sp)    Kanal-Nr.
          move.w    #$40,-(sp)      Funktions-# FREAD
          trap      #GEMDOS         Aufruf
          add.l     #12,sp          Stack berichtigen
          tst.w     d0              Fehlerstatus erfassen
          rts
```

Fortsetzung

GEMDOSF (Fortsetzung)

```
* Pruefen, ob eine Datei lt. FILENAME vorhanden ist
sfirst    clr.w     sattrib         Normale Files suchen
* sfirstl = Einsprung fuer anders vorbesetztes sattrib
sfirstl   bsr       setdta          Setze Disk Transfer Adresse
          move.w    sattrib,-(sp)   Attribut des Files
          move.l    #filename,-(sp) Suchname
          move.w    #$4e,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #8,sp           Stack berichtigen
          tst.w     d0              1=nichts gefunden, 0=etwas gefunden
          rts                       Parameter stehen in DTABUFF

* Folge-File zu sfirst suchen, gleiche Namensbestandteile
snext     move.w    #$4f,-(sp)      Funktionsnummer
          trap      #GEMDOS
          addq.l    #2,sp
          tst.w     d0              01=nichts mehr gefunden, 0=etwas gefunden
          rts                       Name gffs. in FNAMFOUND

* FSEEK positioniert innerhalb einer Datei
* in D0.L muss +- der Offset uebergeben worden sein
fseekbeg  move.w    #0,-(sp)        Suche ab Beginn des Files
          bra.s     fseekl
fseekend  move.w    #2,-(sp)        Suche ab Ende des Files
          bra.s     fseekl
fseek     move.w    #1,-(sp)        Suche ab laufender Position
fseekl    move.l    d0,-(sp)        +- Offset zum Positionieren
          move.w    #$42,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          add.l     #10,sp          Stack berichtigen
          tst.w     d0              Fehler, wenn negativ
          rts

* Pathname zum aktuellen Inhaltsverzeichnis machen
dsetpath  move.l    #filename,-(sp) Zeiger auf den String mit Pathname
          move.w    #$3b,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #6,sp           Stack berichtigen
          tst.w     d0              0=erfolgreich, else Fehler
          rts

* Vollstaendigen Pfadnamen erfassen in PATHBUFF
dgetpath  move.w    #0,-(sp)        Default Aufruf, aktueller Drive
          move.l    #pathbuff,-(sp) Zeiger auf Empfangsbuffer
          move.w    #$47,-(sp)      Funktions-#
          trap      #GEMDOS
          addq.l    #8,sp
          tst.w     d0              0= ohne Fehler
          rts

* GETNSTDH holt vom GEMDOS einen Non-Standard-Handle
* fuer eine System-Device (mit Standard-Handle)
* Der Standard-Handle muss beim Aufruf in D0.W stehen
GETNSTDH  move.w    d0,-(sp)        Standard-Handle aus D0
          move.w    #$45,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #4,sp           Stack berichtigen
          move.w    d0,nstdhdl      Non-Standard-Handle merken
          rts

* SETNSTDH laesst einen Standard-Handle auf dasselbe File
* oder auf dieselbe Device zeigen,
```

Fortsetzung

GEMDOSF (Fortsetzung)

```
* wie den uebergebenen Non-Standard-Handle
* Der Standard-Handle muss in D0.W uebergeben werden
* Der Non-Standard-Handle wird NSTDHDL entnommen
setnsdth  move.w    d0,-(sp)        Standard-Handle in D0
          move.w    nstdhdl,-(sp)   Zuweisung Non-Standard-Handle
          move.w    #$46,-(sp)      Funktions-#
          trap      #GEMDOS         Aufruf
          addq.l    #6,sp           Stack berichtigen
          rts


* Arbeitsspeicher
dfreebuf  ds.l      1               Information Return Buffer/DFREE
totalun   ds.l      1               # Total Allocation Units
bytspsec  ds.l      1               # Bytes per Sector
physsec   ds.l      1               # Physical Sectors/Drive Allocation Unit

bufptri   ds.l      1               Zeiger auf aktuellen Eingabepuffer
bufptro   ds.l      1               Zeiger auf aktuellen Ausgabepuffer
bufleni   ds.l      1               Enthaelt Menge zu lesender Bytes
bufleno   ds.l      1               Enthaelt Menge zu schreibender Bytes
drivemap  ds.l      1               Bitmap der Drives im System
handle    ds.w      1               # des Kanals
handlei   ds.w      1               handle eingabekanal
handleo   ds.w      1               handle Ausgabekanal
nstdhdl   ds.w      1               Non-Standard-Handle
currdrv   ds.w      1               # Current Drive
currdta   ds.l      1               Adresse der aktuellen DTA

sattrib   ds.w      1               File-Attribute

* dtabuff enthaelt 44 Bytes mit File-Angaben
dtabuff   ds.b      21              Unbenutzte Bytes
fattrib   ds.b      1               File-Attribute
ftime     ds.b      2               Zeit der Dateierstellung, binaer
fdate     ds.b      2               Datum der Dateierstellung, binaer
fsize     ds.b      4               Dateigroesse, binaer, Langwort
fnamfound ds.b      14              Gefundener Filename + Extension, Ende DTA

pathbuff  ds.b      64              Pathname mit dgetpath erfasst
filename  ds.b      80              Filename-Arbeitsspeicher
inbuff    ds.b      80              Allg. Eingabepuffer
regsave   ds.l      3               Register-Ablage zur Sicherung
filtflag  ds.w      1               Flag-Variable
adresse   ds.l      1               laufende Adresse merken
lastkey   ds.b      1               Befehlstaste merken
 INCLUDE  GEMDOSFN
 END
```

Kapitel 40

Mit Dateien arbeiten

Die Routinen dieses Abschnittes versetzen den Leser in die Lage, alle wichtigen GEMDOS-Funktionen für die Dateiverwaltung experimentell zum Einsatz zu bringen, um ihre Wirkungen zu studieren. Sie sollen weiterhin als Vorbild für eigene Programmierungen zur Lösung der Massenspeicherung auf Floppy Disk unter Assembler dienen. Beim Studium der Programmabschnitte wird man erkennen, daß die mit GEMDOS gegebenen Lösungsmöglichkeiten einer übersichtlichen Logik folgen.

Frühere Abschnitte dieses Buches brachten bereits das GETLINE-Programm zur gefilterten Eingabe einer Textzeile ("Ein Editor"), einen "Floppy-Test" und die "Floppy-Funktionen", die als dokumentierte Unterprogramme dargestellt wurden. Hier nun fügen wir diese Vorleistungen in einem Menue-Programm zusammen, mit dem eigene Experimente mit Dateien angestellt werden können: Wir können jetzt Ordner und Dateien anlegen, beschreiben und lesen. Die Parameter von Dateien können angezeigt werden, wir können in Inhaltsverzeichnissen suchen und 'Pfadnamen zu Inhaltsverzeichnissen' zu aktuellen Inhaltsverzeichnissen machen.

Zunächst zum Menue-Programm: In der interaktiven Führung soll dem Benutzer auf dem Bildschirm gezeigt werden, welche Auswahlmöglichkeiten er hat. Die Lösung ist mit der PRINTLINE-Funktion unter GEMDOS denkbar einfach. In MAINMENUE wird

das Register D0 auf den Anfang einer sehr langen und mit hexadezimal 00 abgeschlossenen Textkette mit Namen MENUE gesetzt. Innerhalb dieser Textkette sind die einzelnen Zeilen mit den ASCII-Werten der Zeichen CR und LF abgeschlossen, so daß sich bei der Ausgabe jeweils ein Vorschub in eine neue Zeile ergibt. Nach diesem Rezept ist ein Menue schnell programmiert.

Das Menue bringt eine weitere Version zum Thema Fallentscheidungen: In Abhängigkeit von beim Label MAIN1 betätigten Befehlstasten sollen später ab Label FOUND Unterprogramme gezielt mit ihrer Adresse aufgerufen werden. Ab Label MAIN2 lassen wir daher die Register A3 und A5 auf den Beginn der Textketten BEFTAB mit den erlaubten Befehlstasten und auf die Sprungleiste ZIELE zeigen. Nach jedem erfolglos ausgeführten Vergleich müssen wir A3 um 1 auf das nächste Byte weitersetzen und A5 um 1 Langwort (4 Bytes). Das geschieht bei den Labeln MAINLP1 und MAINLP2. – Ist ein Befehl erkannt, so gelangen wir zu FOUND. An dieser Programmstelle zeigt Register A5 noch auf die Adresse des zu wählenden Vektors. Der Vektor selbst muß hier noch in das Register geladen werden, und zwar mit MOVE.L (A5),A5. Der Folgebefehl JSR (A5) ruft dann das Unterprogrammm in der Adressierung 'Adreßregister indirekt' auf.

An der Programmfolge zwischen MAIN2 und FOUND sieht man wiederum, daß Fallentscheidungen mit Programmverzweigung über wenige Befehle sehr einfach implementiert werden können.

Nun zu den ausführbaren Unterprogrammen der Dateiverwaltung: Dem Finden von Dateien im Inhaltsverzeichnis dienen die Funktionen **GFIRST** für ein erstes File mit Namensmerkmalen und **GNEXT** für weitere Files mit gleichem Merkmal. GFIRST verlangt zunächst die Eingabe eines Suchnamens mit JSR GETFNAME. Dieser Dateiname kann ein vollständiger Pfadname sein, in dem auch das Stellvertreterzeichen '?' und der Stern '*' als Kürzel vorkommen dürfen. Dafür Beispiele:

FLOPPY.*
suche eine Datei FLOPPY mit beliebiger Namenserweiterung auf dem aktuellen Laufwerk

FL*.ASM
suche eine Datei mit Namensbeginn FL und der Erweiterung .ASM

C: \ ORDNER \ FLOPPY?.ASM
suche auf dem Laufwerk C (z. B. RAMDISK) im Ordner ORDNER nach einer ersten Datei, auf deren Namen FLOPPY noch ein weiteres Zeichen folgt und die die Erweiterung .ASM hat.

Die Funktion GFIRST bringt einen so gefundenen Dateinamen zusammen mit den Dateiparametern zu Anzeige. Das sind das Attribut Lesen/Schreiben, die Daten der Dateierstellung und die Dateigröße. Wenn keine Datei entsprechend dem Suchkriterium gefunden wurde, so erfolgt bei NONAME ein textlicher Hinweis. – GNEXT setzt die Suche nach weiteren Dateien mit dem Kriterium fort.

GFIRST und GNEXT kann man in ein Menueprogramm einbinden, das dem Benutzer die Möglichkeit bietet, eine gefundene Datei zu benutzen. In diesem Fall ist der gefundene Dateiname FNAMFOUND nach FILENAME zu übertragen, damit die Funktion GOPEN benutzt werden kann. Beim Lesen wird man auch die Dateigröße in der Variablen FSIZE berücksichtigen, um das Ende einer Datei zu erkennen.

GETFREE bringt die sog. Allocation Units zur Anzeige, aus denen man für ein Laufwerk errechnen kann, wieviele Bytes noch frei sind. Die Formel dafür ist:

(Sektoren je Allocation Unit) * (Bytes je Sektor) * (freie Allocation Units).

Man benutzt diese Funktion vor einer Abspeicherung, um zu sehen, ob der Platz noch reicht.

GCREATRW legt eine Datei erstmals an. Es wird ihr Name (Pfadname) abgefordert, womit sie auf beliebigen vorhandenen Laufwerken und in beliebigen Ordnern erscheint. Diese Funktion wurde für eine Schreib-/Lesedatei geschrieben. Für Dateien mit anderen Attributen ist analog zu verfahren. – Mit der Anlage wird die Datei geöffnet (sie ist später mit GCLOSE zu schließen), und wir erhalten eine Kanal-Nummer (Handle) für die Funktionen FREAD, FRITE oder FCLOSE. Eine so geschaffene Datei ist noch leer, erscheint aber schon im Inhaltsverzeichnis. – Wenn der gewählte Name mit einem schon vorhandenen gleich ist, so wird der Inhalt der vorhandenen Datei gelöscht. – Mit **SHOWHAND** kann der zu einer Datei erhaltene Handle zur Anzeige gebracht werden.

Um die Anlage einer Datei erproben zu können, wird mit der Befehlstaste 'b' das Unterprogramm TESTFILE aufgerufen. Es verlangt vom Benutzer die Angabe eines Namens. Danach schreibt es die Bytes zwischen MENUE und MENUEND (das ist der Längenparameter für die GEMDOS-Funktion FWRITE) auf das Laufwerk und schließt die Datei. – Damit ist nach dem Lesen einer Datei mit FREAD (Abschnitt "Floppy-Test") nun auch das Prinzip des Schreibens geregelt. Wesentlich für FWRITE ist eigentlich nur die Angabe, wieviele Bytes geschrieben werden sollen. Der Benutzer richtet das nach seinen Bedürfnissen ein. Wenn ein Textdokument abgespeichert werden soll, so wird man im allgemeinen seinen gesamten schon beschriebenen Puffer abspeichern. Wenn man daraus Ausschnitte konservieren möchte, so wird man sich entsprechende Marken setzen.

Bei allen Tätigkeiten, die etwas mit laufender Aufzeichnung z. B. für Buchungssätze oder Meßwerte zu tun haben, wird man einen ausreichend großen Empfangspuffer für solche Daten vorsehen und jeweils den Pufferinhalt auf die Datei absetzen.

Die **FSEEK**-Funktion zum Positionieren innerhalb einer Datei wurde hier nicht ausprogrammiert. Mit ihr kann man nicht nur Zugriff auf strukturierte Dateien nehmen (positioniere auf den Datensatz X), sondern z. B. auch eine vorhandene Datei erweitern (Append-Funktion am Dateiende).

GMKDIR legt einen Ordner an, ein Unter-Inhaltsverzeichnis. Es wird zunächst der Name abgefragt, der z. B. wie folgt eingegeben wird A:\MYFILES. Im Inhaltsverzeichnis des Laufwerkes A werden wir danach einen leeren Ordner mit Namen MYFILES sehen. In diesen können wir später Dateien kopieren oder schreiben. – **GSETPATH** macht den Inhalt eines Ordners zum aktuellen Inhaltsverzeichnis. Auf die Namensanforderung geben wir für den vorstehenden Fall ein: A: \ MYFILES \ . Damit beziehen sich weitere Dateiverarbeitungen jetzt auf das Inhaltsverzeichnis unter MYFILES (Suchen, Lesen, Schreiben), und wir müssen nicht immer dem Dateinamen den Pfadnamen voranstellen. – **GGETPATH** zeigt den aktuellen Pfadnamen an. Diese Funktion dürfte aber keinen so großen praktischen Nutzen haben.

Mit **GGETDTA** erhält man von GEMDOS einen Zeiger, unter dem man einen Disk Transfer Buffer einrichten kann, den man mit GSETDTA aktiviert. Die hier abgedruckten Routinen arbeiten bei den Anzeigen mit einem eigenen DTABUFF. – **GGETDRIV** und **GSETDRIV** zeigen die Nummer des aktiven Laufwerkes bzw. setzen sie. Dabei haben die Laufwerke A ... P die Hex-Nummern 0 ... F.

Zum Schluß sind noch zwei Hilfsprogramme zu erwähnen, die auch an anderer Stelle gute Dienste erledigen können: **GETADRES** holt einen ASCII-String von der Tastatur ein und konvertiert ihn zu binär im Register D1 und mit Ablage bei ADRESSE, soweit in ihm ab Beginn verwertbare Hexziffern enthalten sind. Die Umwandlung bricht beim ersten Nicht-Hexzeichen ab. Andererseits wird ab 9. Eingabeziffer wieder links aus dem Register D1 hinausgeschoben. – Mit **LONGOUT, WORDOUT** und **BYTEOUT** werden Langworte, Worte oder Bytes, auf die das Register A5 weist, als Hexziffern ausgegeben.

FLOPPY 2

```
* FLOPPY2
* Anwendung der GEMDOS-Floppy-Funktionen
* unter Rueckgriff auf frueher erklaerte Unterprogramme

cr        equ       $0d            Carriage Return
lf        equ       $0a            Linefeed

 SECTION  ONE

main      movem.l   a3-a5,regsave  Register sichern
mainmenue move.l    #menue,d0      Menue anzeigen
          jsr       printline
mainl     jsr       dircon         Taste holen
          move.b    d0,lastkey
          jsr       conout         und anzeigen
          jsr       crlf
main2     lea       beftab,a3      Zeiger auf Befehlstasten
          lea.l     ziele,a5       Zeige auf den ersten Vektor der Sprungtabelle
          move.b    lastkey,d0     Taste zurueck
mainloop  tst.b     (a3)           Ende der Befehlstabelle?
          beq.s     mainl          ja, Taste nicht gueltig
mainlp1   cmp.b     (a3)+,d0       Taste : Befehle
          beq.s     found          Gueltiger Befehl erkannt
mainlp2   adda.l    #4,a5          Auf Folge-Vektor zeigen
          bra.s     mainloop       Weitersuchen
found     move.l    (a5),a5        Vektor aus Zeigeradresse entnehmen
          jsr       (a5)           Vektorierte Routine ausfuehren
          bra.s     mainmenue      Zur Befehlsebene zurueck

abbruch   adda.l    #4,sp          Stack wegen RTS berichtigen
          movem.l   regsave,a3-a5  Register zurueckholen
          jmp       term           Programm beenden

* Programmstummel fuer nicht ausgefuehrtes Unterprogramm
return    rts

* Zeige den aktuellen Handle (Kanal) an
showhand  lea.l     handle,a5      Zeiger auf das Wort
          jsr       wordout
          jsr       crlf
          rts

* GMKDIR legt einen Ordner gemaess Pfadnamen an
gmkdir    jsr       getfname       Pfadnamen holen
          jsr       mkdir          Ordner anlegen
          rts

* Pfadnamen zum aktuellen Inhaltsverzeichnis machen
gsetpath  jsr       getfname       Pfadnamen vorgeben
          jsr       dsetpath       Aktuell machen
          rts

* Aktuellen Pfadnamen holen und anzeigen
ggetpath  jsr       dgetpath       Aktuellen Pfadnamen holen
          move.l    #pathbuff,d0   Zeiger auf String setzen
          jsr       printline      und anzeigen
          jsr       crlf
          rts

* Disk-Free Parameter holen und anzeigen
getfree   jsr       dfree          Funktionsaufruf, Parameter holen
          lea.l     dfreebuf,a5    Zeiger auf den Buffer setzen
          jsr       longout        Freie Allocation Units, binaer anzeigen
```

Fortsetzung

FLOPPY 2 (Fortsetzung)

```
          jsr       spaceout
          jsr       longout        Allocation Units insgesamt
          jsr       spaceout
          jsr       longout        Bytes pro Sektor (512=$200)
          jsr       spaceout
          jsr       longout        Sektoren pro Allocation Unit (2)
          jsr       crlf
 jmp mainloop
          rts

* Hole Nummer des aktuellen Laufwerkes und zeige sie an
ggetdriv  jsr       getdrv         Funktionsaufruf
          lea.l     currdrv,a5     Zeiger auf Nummer
          jsr       wordout        Als Langwort ausgeben
          jsr       crlf
          rts

* Setze eine Laufwerksnummer als die aktuelle
gsetdriv  jsr       getadres       Hexadezimale Nummer im String holen
          clr.w     d0             Register =00
          or.b      d1,d0          Nummer uebertragen
          jsr       setdrv         Funktionsaufruf
          rts

* Setze die mit GGETDTA geholte DTA-Adresse als die aktuelle ein
gsetdta   move.l    #currdta,-(sp) Zeiger auf Puffer
          jmp       setdtal        Fortsetzung und RTS dort

* Hole eine Disk Transfer-Adresse (DTA) und zeige sie an
ggetdta   jsr       getdta         Funktionausruf
          lea.l     currdta,a5
          jsr       longout        Ausgabe der DTA-Adresse
          jsr       crlf
          rts

* Schreibe des Menue dieses Programmes als eine Datei
testfile  jsr       gcreatrw                  File mit Namen anlegen
          move.l    #menue,bufptro            Zeiger auf Anfang
          move.l    #menuend-menue,bufleno    Laengenparameter
          jsr       fwrite                    Datei schreiben
          jsr       fclose                    und wieder schliessen
          rts

* Lege ein Schreib-/Lesefile mit Namen an
gcreatrw  jsr       getfname       Dateinamen anfordern
          jsr       createrw       Datei anlegen, Lesen/Schreiben
          rts

* Oeffnen einer Schreib-/Lesedatei mit Namen
gopen     jsr       getfname       Dateinamen anfordern
          jsr       fopenrw        Oeffne Datei
          rts

gclose    jmp       fclose         Datei schliessen, RTS dort

* Dateinamen anfordern und erste entsprechende Datei suchen,
* ihre Parameter aus der DTA anzeigen
gfirst    jsr       getfname       Dateinamen anfordern
          jsr       crlf           Neue Zeile
          jsr       sfirst         Normale R/W-Dateien suchen
gfirst1   bne.s     noname         Nichts (mehr) gefunden
gfirst2   move.l    #fnamfound,d0  Zeige auf gefundenen Namen
          jsr       printline      Anzeige
```

Fortsetzung

FLOPPY 2 (Fortsetzung)

```
          jsr       crlf
          lea.l     fattrib,a5      File-Attribut zeigen,
          jsr       byteout         alles binaer
          jsr       spaceout
          lea.l     ftime,a5        Zeit der Erstellung anzeigen
          jsr       wordout
          jsr       spaceout        Space dahinter
          lea.l     fdate,a5        Dito Datum
          jsr       wordout
          jsr       spaceout
          lea.l     fsize,a5        Dateigroesse
          jsr       longout
          jsr       spaceout
          jsr       crlf
          clr.w     d0              BEQ-Status setzen
          rts
* Suche weitere Files mit den Vorgaben von GFIRST
gnext     jsr       snext           Funktionsaufruf
          bra.s     gfirstl         Anzeigen wie bei GFIRST

* Anzeige, dass keine Datei lt. Vorgaben mehr folgt
noname    move.l    #text8,d0       Meldung: Nichts gefunden
          jsr       printline
          jsr       crlf
          move.w    #$ffff,d0       Status negativ
          rts

* Eine Adresse als ASCII-String anfordern und nach binaer konvertieren
getadres  move.l    #text2,d0       Adresse anfordern
          jsr       printline
          clr.l     d1              Loesche Adressfeld in D1
          lea.l     inbuff,a5       Allgemeinen Buffer benutzen
          jsr       getline         String mit Zahlen holen
          lea.l     inbuff,a5       Wieder auf den Anfang setzen
getadrl   move.b    (a5)+,d0        Zeichen fuer Zeichen holen
          jsr       hexfilt         Hexzahl?
          bmi.s     getadrz         nein, Abbruch bei sonstigen Zeichen
          cmpi.b    #$41,d0         Ziffern A...F, a...f
          bcs.s     getadr2         nein
          addq.b    #7,d0           Hexwert korrigieren
getadr2   andi.l    #$000f,d0       maskieren
          move.w    d0,-(sp)        Ziffer soweit merken
          move.w    #4,d0           Verschiebung vorbereiten
          asl.l     d0,d1           In D1 4* links schieben
          move.w    (sp)+,d0        Ziffer zurueck
          or.l      d0,d1           Nach D1 hinzu-odern

          bra.s     getadrl         Naechste Ziffer konvertieren
getadrz   move.l    d1,adresse      Adressenbildung merken
          jsr       crlf            Neue Zeile
          rts


* LONGOUT, WORDOUT und BYTEOUT
* bringen den Inhalt beliebiger Langwoerter, Woerter, Bytes
* zur Bildschirmausgabe. Der Adresszeiger muss im Register A5
* vor dem Aufruf uebergeben worden sein.
* Es wird die Routine HEXOUT des Abschnittes Memory-Dump benutzt
longout   bsr.s     byteout         Vorderstes Byte ausgeben
          bsr.s     byteout         2. Byte des Langwortes
wordout   bsr       byteout         1. Byte des Wortes
byteout   move.b    (a5)+,d0        Ein Byte ausgeben
          jmp       hexout          RTS bei HEXOUT
```

Fortsetzung

FLOPPY 2 (Fortsetzung)

```
spaceout  move.b    #$20,d0        1 Zwischenraumzeichen
          jsr       conout         ausgeben
          rts

* GETFNAME fordert zur Eingabe eines Datei- oder Pfadnamens auf
getfname  jsr       crlf           Neue Zeile
          move.l    #text1,d0      Zeiger auf interaktiven Text setzen
          jsr       printline      Ausgabe per GEMDOS
          lea.l     filename,a5    Benutze Namens-Puffer fuer die Eingabe
          jsr       getline        1 Zeile holen
          cmpi.b    #$1b,d0        Wurde ESCAPE zum Abbruch gegeben?
          rts

* Sprungadressen der Unterprogramme
ziele     dc.l      gopen,fclose,gcreatrw,gsetdriv
          dc.l      ggetdriv,ggetdta,gsetdta,getfree
          dc.l      gmkdir,fread,testfile,gfirst
          dc.l      gnext,return,gsetpath,ggetpath
          dc.l      showhand,abbruch
* Erlaubte Befehlstasten
beftab    dc.b      '123456789abcdefgh',03,0

* Textkonstante
text1     dc.b      'Eingabe Filename/Pathname:   ',0
text2     dc.b      'Adresse eingeben:  ',0
text3     dc.b      'Handle ist  ',0
text4     dc.b      'Free / total Allocation Units bytes/sect sect/Alloc',0
text5     dc.b      'Gefundener Dateiname ist:  ',0
text6     dc.b      'Pfadname ist:  ',0
text7     dc.b      'Aktuelle DTA ist:  ',0
text8     dc.b      'Nichts (mehr) gefunden',0

menue     dc.b      '  Befehlsauswahl mit Tastendruck',cr,lf
          dc.b      '  Abbruch mit CTRL-C',cr,lf
          dc.b      '1  FOPEN Name',cr,lf
          dc.b      '2  FCLOSE',cr,lf
          dc.b      '3  Create a File R/W',cr,lf
          dc.b      '4  Set Drive 0...15',cr,lf
          dc.b      '5  Get Drive',cr,lf
          dc.b      '6  Get Disk Transfer Address',cr,lf
          dc.b      '7  Set Disk Transfer Address',cr,lf
          dc.b      '8  Zeige Disk Free Space',cr,lf
          dc.b      '9  Ordner anlegen',cr,lf
          dc.b      'a  Lesen von Floppy',cr,lf        *
          dc.b      'b  Schreiben auf Floppy',cr,lf
          dc.b      'c  Suche SFIRST',cr,lf
          dc.b      'd  Suche SNEXT',cr,lf
          dc.b      'e  Positionieren FSEEK',cr,lf     *
          dc.b      'f  DSETPATH',cr,lf
          dc.b      'g  DGETPATH',cr,lf
          dc.b      'h  Zeige Handle',cr,lf
          dc.b      'CTRL-C = Programmabbruch',cr,lf,0
menuend                          ;Ende des Menues

* Arbeitsspeicher
dfreebuf  ds.l      1              Information Return Buffer/DFREE
totalun   ds.l      1              # Total Allocation Units
bytspsec  ds.l      1              # Bytes per Sector
physsec   ds.l      1              # Physical Sectors/Drive Allocation Unit

bufptri   ds.l      1              Zeiger auf aktuellen Eingabepuffer
bufptro   ds.l      1              Zeiger auf aktuellen Ausgabepuffer
bufleni   ds.l      1              Enthaelt Menge zu lesender Bytes
```

Fortsetzung

FLOPPY 2 (Fortsetzung)

```
bufleno   ds.l     1          Enthaelt Menge zu schreibender Bytes
drivemap  ds.l     1          Bitmap der Drives im System
handle    ds.w     1          # des Kanals
handlei   ds.w     1          handle eingabekanal
handleo   ds.w     1          handle Ausgabekanal
nstdhdl   ds.w     1          Non-Standard-Handle
currdrv   ds.w     1          # Current Drive
currdta   ds.l     1          Adresse der aktuellen DTA

sattrib   ds.w     1          File-Attribute

* dtabuff enthaelt 44 Bytes mit File-Angaben
dtabuff   ds.b     21         Unbenutzte Bytes
fattrib   ds.b     1          File-Attribute
ftime     ds.b     2          Zeit der Dateierstellung, binaer
fdate     ds.b     2          Datum der Dateierstellung, binaer
fsize     ds.b     4          Dateigroesse, binaer, Langwort
fnamfound ds.b     14         Gefundener Filename + Extension, Ende DTA

pathbuff  ds.b     64         Pathname mit dgetpath erfasst
filename  ds.b     80         Filename-Arbeitsspeicher
inbuff    ds.b     80         Allg. Eingabepuffer
regsave   ds.l     3          Register-Ablage zur Sicherung
filtflag  ds.w     1          Flag-Variable
adresse   ds.l     1          laufende Adresse merken
lastkey   ds.b     1          Befehlstaste merken
 INCLUDE  GEMDOSFN
 END
```

Kapitel 41

Serielle Datenübertragung

Die serielle Schnittstelle wird mittels der GEMDOS-Routine **AUXIN** benutzt, um eine Textdatei von einem anderen Computer oder Terminal in den Speicher des Atari zu übertragen. Hier kann die Datei zeilenweise betrachtet, ergänzt und schließlich auf Floppy Disk abgespeichert werden. Unter Heranziehung von Routinen, die in anderen Buchabschnitten aufgebaut wurden, ist damit ein komplettes Anwenderprogramm zur Datenübertragung gegeben.

Überblick. Vom Desktop-Programm des Atari aus haben wir die Möglichkeit, den **VT52-Emulator** für den seriellen Datenverkehr mit dem Rechner zu benutzen, so daß dieser als Terminal dient. Empfangene Information wird dabei nicht gespeichert. Andererseits kann der Inhalt von Dateien mühelos auf andere Rechner übertragen werden, wenn wir ihr Bild auf dem Desktop zweimal kurz anklicken, um sie zu öffnen. Im nun erscheinenden Antwortkasten können wir das Drucken der Datei wählen. Wenn wir zuvor die Druckeranpassung auf MODEM vorgenommen hatten, wird der Inhalt der Datei seriell abgesendet.

Das Programm RS232 der nachfolgenden Liste schließt die bisher bestehende Lücke im Grundprogramm des Systems, daß man nämlich empfangene Dateien nicht zwischenspeichern und dann auf Floppy Disk absetzen konnte. Damit wird es möglich, z. B. Quelltexte für Programme von anderen Rechnern zu empfangen, sie in einem Editor des Atari zu bearbeiten und sie dann

seinen Assemblern oder Sprachkompilern zur Übersetzung zu unterwerfen. Analog gilt das für Texte schlechthin.

Die Grundidee ist, ein Programm wie in "Ein Editor" beschrieben, derart zu benutzen, daß die Eingabe nicht von der Tastatur her mit dem Unterprogramm GETLINE erfolgt, sondern vom seriellen Kanal mit der Routine **AUXIN** (Auxiliary Input von der ACIA). Empfangene Zeichen werden sofort in das RAM abgespeichert. An ein empfangenes Carriage Return wird automatisch ein Linefeed angehängt, ob sich letzteres nun im Übertragungstrom befand oder nicht.

Als Zeichen für das Ende der Übertragung wird CTRL-Z angesehen. Andere Zeichen oder Zeichenfolgen kann man je nach Übertragungsformat leicht vorsehen und ggfs. interaktiv abfragen. Wenn das Ende der Übertragung erkannt ist, gelangt man in eine kleine Befehlsebene bei MAINCMD. Hier hat man die Möglichkeit, mit '**t**' auf die oberste Zeile einzustellen und dann mit '**d**' (down) den empfangenen Text zeilenweise zur Kontrolle zu betrachten. Mit '**a**' kann man weiteren Text empfangen und an das bisher erreichte Textende anhängen.

Die Abspeicherung auf die Floppy geschieht mit der Befehlstaste '**f**' und erfolgt ab Label FLOPPY. Hier werden die im Abschnitt "Mit Dateien arbeiten" entwickelten Unterprogramme zum Anlegen einer Datei mit Namen (GCREATRW), zum Schreiben (FWRITE) und zum Schließen (FCLOSE) zum Einsatz gebracht. Erwähnenswert ist, das FWRITE einen Zeiger auf den Beginn des Textpuffers und eine Angabe über die Menge der zu schreibenden Bytes benötigt. Diese Parameter werden mit MOVE.L #EDITOR,BUFPTR0 und der nachfolgenden Differenzbildung nach BUFLEN0 übergeben.

Zum Programmaufbau: Im Abschnitt nach MAIN mit der Meldung des Programmes wird abgefragt, ob in der absendenden Datei als DIN-ASCII codierte deutsche Sonderzeichen für Zwecke der Darstellung auf dem Atari umgewandelt werden sollen. Wenn ja, dann erfolgt diese Umwandlung nach dem Zei-

chenempfang bei MLOOP in der folgenden DINLOOP. Der Zeiger in Register A4 gleitet dabei mit (A4)+ über eine Tabelle ASCIITAB. Wird ein Sonderzeichen erkannt, so wird seine Umcodierung mit dem Befehl MOVE.B (A4,A5.W),d0 aus der DINTAB entnommen; die Adressierungsart "Adreßregister indirekt mit Index (in A5) mit Offset 0" dieses Befehls erlaubt ein schnelles Aufschlagen in Tabellen. – Mit dieser Methode kann man natürlich auch andere Sonderzeichen bequem umsetzen, wenn man die Zuordnungstabellen entsprechend wählt. Bei größeren Tabellen zur Umcodierung wird man allerdings den ASCII-Wert des empfangenen Zeichens als Index in eine Tabelle benutzen.

Zum Empfang von Zeichen über die RS232-Schnittstelle noch folgende Hinweise: Bei zahlreichen Versuchen zeigte es sich, daß der Atari im Tempo nicht mithält, wenn er Zeichen in einem "Trommelfeuer" mit 9600 Baud empfangen und auf Bildschirm auch noch darstellen soll. Offensichtlich ist die Bildschirmbedienung dafür nicht schnell genug. Es wurde daher auf eine sofortige Darstellung empfangener Zeichen verzichtet. Weiterhin hat es sich als zweckmäßig erwiesen, bei MLOOP zunächst einmal mit JSR AUXINSTAT zu prüfen, ob überhaupt ein Zeichen vorliegt, ehe man es auszulesen versucht.

RS 232

```
* RS232
* Eine Textdatei seriell empfangen und auf Floppy abspeichern
* Programmabbruch mit CTRL-C

 SECTION  ONE
main      move.l    #textr,d0       Zeiger auf interaktiven Text setzen
          jsr       printline       Ausgabe per GEMDOS
          jsr       crlf
          clr.b     dinflag         Umlaute nicht konvertieren
          movem.l   a3-a5,regsave   Register sichern
          move.l    #textdin,d0     Umwandlung von Sonderzeichen abfragen
          jsr       printline
          jsr       conin           Tastendruck? mit Echo
          move.w    d0,-(sp)        Zeichen merken
          jsr       crlf            Neue Zeile, veraendert D0
          move.w    (sp)+,d0        Zeichen zurueck
          cmp.b     #'j',d0         j=JA?
          bne.s     main0
          not.b     dinflag         nein, dann Flag negativ machen
main0     lea       editor,a3       Zeige auf Beginn des Textspeichers
          move.l    a3,edipntr      Einsatzpunkt merken
          clr.b     (a3)            Dort eine Null als Begrenzer ablegen
mloop     jsr       auxinstat       Liegt Zeichen vor an RS232?
          beq.s     mloop           nein
mloop0    jsr       auxin           Hole Zeichen vom RS232-Kanal
          and.w     #$007f,d0       Zeichen zu ASCII machen
          cmp.b     #$1a,d0         Abbruch mit CTRL-Z?
          beq.s     rcvend          ja
          cmp.b     #$0a,d0         Linefeed nicht doppelt ablegen
          beq.s     mloop
          tst.b     dinflag         Umlaute konvertieren?
          beq.s     mloop1          nein, dann ablegen
          lea       asciitab,a4     Zeige auf ASCII-Tabelle
          move.w    #-1,a5          Mitlaufender Zaehler A5=$ffff
dinloop   adda.w    #1,a5           Zaehler+1
          tst.b     (a4)            Ende der Tabelle?
          beq.s     mloop1          ja
          cmp.b     (a4)+,d0        Ist das Zeichen in der Tabelle?
          bne.s     dinloop         nein
          lea       dintab,a4       Konversionstafel
          move.b    (a4,a5.w),d0    Neues Zeichen entnehmen
mloop1    move.b    d0,(a3)+        Zeichen im Editor ablegen
          cmp.b     #$0d,d0         CR als Zeilenbegrenzer?
          bne.s     mloop           nein
          move.b    #$0a,(a3)+      Linefeed im Editor anhaengen
          bra.s     mloop

* Ende der Uebertragung erkannt
rcvend    clr.b     (a3)            00 als Begrenzer anhaengen
          move.l    a3,edipntr      Einsatzpunkt merken

* Befehlsebene
maincmd   jsr       crlf            Neue Zeile
          move.l    #text2,d0       Befehlsebene anzeigen
          jsr       printline
maincmd1  jsr       conin           Antwort holen mit Echo
          move.w    d0,-(sp)        Taste auf dem Stack merken
          jsr       crlf
          move.w    (sp)+,d0        Tastaturzeichen zurueck
          cmp.b     #'a',d0         Append Text ans Ende?
          beq.s     append          ja
          cmp.b     #$03,d0         CTRL-C als Abbruch?
          beq.s     abbruch         ja
```

Fortsetzung

RS 232 (Fortsetzung)

```
          cmp.b     #'t',d0          Auf Topzeile setzen?
          beq.s     topzeile         ja
          cmp.b     #'d',d0          Down, in Folgezeile setzen?
          beq.s     down             ja
          cmp.b     #'f',d0          Auf Floppy speichern?
          beq       floppy           ja
          bra.s     maincmd          Kein gueltiger Befehl

abbruch   movem.l   regsave,a3-a5    Register zurueckholen
          jmp       term

append    move.l    edipntr,a3       Alter Stand
          bra       mloop            Weitere Zeichen holen
topzeile  lea.l     editor,a3        Zeiger auf Textbeginn
down      tst.b     (a3)             Sind wir am Ende des Editors?
          beq.s     maincmd          ja, neuen Befehl holen
          jsr       showline         Zeile anzeigen
          bra.s     maincmd          Weiteren Befehl mit Prompt

floppy    jsr       gcreatrw         Lege Datei an
          move.l    #editor,bufptro Zeige auf Pufferbeginn fuer fwrite
          move.l    edipntr,d0       Ende des benutzten Speichers
          addq.l    #1,d0            edipntr zeigte auf den Begrenzer
          sub.l     #editor,d0        minus Beginn-Adresse
          move.l    d0,bufleno       = Menge Bytes
          jsr       fwrite           GEMDOS: Schreiben auf Floppy
          jsr       fclose
          bra       maincmd          Naechster Befehl

* Arbeitsspeicher
textl     dc.b      'Eingabe Filename/Pathname:   ',0
textr     dc.b      'RS232-Programm',0
text2     dc.b      'Befehl eingeben: a t d f CTRL-C  ',0
textdin   dc.b      'Umwandlung von Umlauten  j/n?  ',0
asciitab  dc.b      $5b,$5c,$5d,$7b,$7c,$7d,$7e  DIN-ASCII
dintab    dc.b      $8e,$99,$9a,$84,$94,$81,$9e  ATARI-Code

dinflag   ds.b      1                Soll konvertiert werden?
editor    ds.b      128*1024
ediende                              ;Label der Bereichsbegrenzung
```

RECHENROUTINEN UND DATENFELDER

RECHENROUTINEN UND DATENFELDER

Kapitel 42

Binäres Rechnen

Addition, Subtraktion, Multiplikation und Division sind auf dem 68000 vor allem für binäres Rechnen implementiert, wobei wir bei der Strichrechnung die Operandenbreiten Byte, Wort und Langwort benutzen können. Für die Punktrechnungen gelten besondere Regeln. Bei einigen Befehlen muß wenigstens ein Operand in einem Datenregister stehen, worauf bei der Programmierung zu achten ist. Für dezimales Rechnen, das in einem anderen Abschnitt abgehandelt wird, stehen nur Addition und Subtraktion zur Verfügung, und außerdem nur byte-weise. Aus Gründen der schnelleren Ausführung wird es sich daher im allgemeinen empfehlen, Zahlen intern binär zu speichern und von den binären Rechenarten Gebrauch zu machen. Die Frage der Zahlenumwandlung tritt dann nur bei der Ein- und Ausgabe auf. Das nachstehende Programm behandelt alle vier Grundrechenarten und sorgt auch für die Umwandlung der Eingabe nach binär bzw. hex. An verschiedenen Stellen, bei Verschiebebefehlen und beim Rechnen, wird die Behandlung des Extendbits im Status gezeigt.

Ziel unserer Programmierung ist es, unter Mitbenutzung früher entwickelter Routinen wie **PRINTLINE, GETLINE, HEXFILT, HEXOUT** usw. einen hexadezimalen Tischrechner zu entwickeln. Bei Addition und Subtraktion kann er jeweils zwei Zahlen von 64 Bit, d. h. von 16 hexadezimalen Ziffern, zu einem Ergebnis verknüpfen; bei der Multiplikation von 32 x 32 Bit zu einem Ergebnis von

64 Bit. die etwas kompliziertere Division ist in 32 / 16 Bits implementiert.

Rechenaufgaben werden zeilenweise mit GETLINE eingegeben. Dieses Unterprogramm gestattet zwar die Eingabe beliebiger druckbarer Strings; was als Eingabe zugelassen ist, unterliegt jedoch für dieses Beispiel einer etwas strengen Syntax: Nach dem Label MAIN1 wird zunächst eine Zeile unbesehen eingelesen. Bei MAIN1A wird das Adreßregister A5 wieder auf den Zeilenbeginn gerichtet, und es wird Zeichen für Zeichen aus dem Eingabepuffer geholt. Eine Zeile darf dabei leer sein (an ihrem Beginn steht 00) oder sie darf mit führenden Zwischenräumen beginnen. Danach muß mindestens eine Hexziffer folgen, dahinter (nur) ein Zwischenraum (Space), dann einer der Operatoren +, –, *, / für die Rechenart, wieder (nur) ein Space und dann der zweite Operand. Diese Syntax in einem Beispiel:

A501 * 3333 (Multipliziere $A501 mit $3333)

Wir verlangen nicht, daß Zahlen mit führenden Nullen geschrieben werden, wenn es sich um kleine Zahlen handelt. Wir werden allerdings auf einen Überlauf aufmerksam gemacht, wenn für einen Operanden mehr als 8 Ziffern (32 Bits) eingegeben wurden.

Umwandlung der ASCII-Eingabe. Die Umwandlung der mit GETLINE eingegebenen ASCII-Zeichen in Binärzahlen erfolgt mit dem Programm **CONV** (wie convert, wandle um). Es hat aus Gründen der Darstellung, die man auch anders lösen kann, zwei Äste der Bearbeitung: Die erste eingegebene Zahl wird in die zwei Langworte im Speicher bei OPER1H umgewandelt (der höherwertige Teil steht "links"), die zweite Zahl wird in den Datenregistern D2 (High) und D3 (Low) gebildet. Die Umsteuerung erfolgt mit der Variablen CFLAG, die anfangs 00 ist, im zweiten Durchgang "NOT CFLAG", also $FF.

Innerhalb des Unterprogrammes CONV wird auf Hexziffern gefiltert. Am Ende eines Zahlenstrings oder bei unerlaubten Einga-

ben wird die Umwandlung abgebrochen, und man kehrt in die Ebene von MAIN zurück.

Hinweise zum Umwandlungsprogramm CONV: In der ASCII-Darstellung folgen die Ziffern A ... F nicht unmittelbar auf 0 ... 9. Daher sind Buchstaben (hex 41 ... 46) durch Addition von 9 auf Hexwerte in den unteren 4 Bits zu bringen (Zeilen vor dem Label CONV1). Dabei entstehen dort die binären Muster 1010 ... 1111. Bei CONV1 können dann die Bits 4 ... 7 der Eingabe abmaskiert werden. – Die Eingabezeile wird von links nach rechts gelesen, die höchstwertigen Ziffern der Operanden werden daher zuerst erfaßt. In der Umwandlung bedeutet es, daß wir das aufgelaufene Zwischenergebnis mit jeder ausgewerteten Hexziffer um 4 Bitpositionen nach links rücken müssen. Hier treten nun einige Bedenken ein: Das Ergebnisfeld OPER1H (bzw. das Registerfeld D2/D3) hat die Breite von 2 Langworten = 64 Bits.

Verschiebebefehle. Wie überträgt man links vorne herausgeschobene Bits rechts hinten in das vorangehende, höherwertige Langwort? Aus der Befehlsdokumentation ist zu ersehen, daß der benutzte Befehl **ASL** das links herausrutschende Bit in das Extendbit des Bedingungscode-Registers schibt. Wenn wir nun bei der Linksverschiebung des höherwertigen Langwortes den Befehl **ROXL** (verschiebe links mit Einzug des Extendbits) benutzen, dann wird über 2 Langworte hinweg richtig verschoben.

Verschiebungen von Speicherinhalten können jeweils nur um 1 Bitposition erfolgen; der Operand darf immer nur jeweils 1 Datenwort sein. Wir bauen daher um das Label CONV2 eine viermal zu wiederholende Schleife mit dem Konstrukt 'Decrement and Branch' auf (vermindere das zählende Datenregister D1.W; wenn es noch nicht negativ geworden ist, verzweige zurück zu CONV2). Innerhalb der Schleife wird jedes der 4 zu berücksichtigenden Speicherworte um jeweils 1 Bitposition nach links verschoben; erster Befehl ist ASL, dann dreimal ROXL.

Es mag der Wunsch aufkommen, die notwendigen Verschiebungen schneller zu bewirken, indem man die Verschiebungen

in Datenregistern vornimmt, so wie ab Label CONV4. Bei Datenregistern kann man sogar mit einem Direktoperanden angeben, um wieviele Bitpositionen mit einem Befehl geschoben werden soll. Letztere Adressierungsart ist hier aber nicht angepaßt, weil es keinen mehrstelligen Zwischenspeicher vom Typ des Extendbits gibt, aus dem man dann mühelos in den höherwertigen Teil des Operanden einziehen könnte. Bei ASL #x,Dy (verschiebe im Datenregister y um x Positionen nach links) zeigt das V-Flag im Status nur an, ob sich das vorderste Bit überhaupt einmal änderte. Wir müssen daher auch bei CONV4 für die zweite Eingabezahl bitweise nach links verschieben, können hier aber ganze Datenregister mit 32 Bit benutzen.

Zurück zum Hauptprogramm in die auf Label MAIN2 folgenden Befehle: Nach der im Grunde sehr einfachen Umwandlung des ersten Zahlenstrings von ASCII nach Hex folgen syntaktische Abfragen, ob ein Space, ein Operator und ein weiterer Space folgen. Bei MAIN2A ist der zweite Zahlenstring erreicht, der mit CONV umgewandelt wird, und zwar jetzt in zwei Datenregister. Je nach Operator, der im Byte OPART gemerkt wurde, gelangt man jetzt nach den Abfragen zu einer der ausführenden Programmteile **ADDIT**, **SUBIT**, **MULIT** oder **DIVIT**. Hierzu folgende Ausführungen:

Addition und Subtraktion. ADDIT addiert zwei Langworte in den Datenregistern D2/D3 zu den zwei Langworten im Speicher bei OPER1H. Der erste benutzte Befehl ADD ist in seinen möglichen Adressierungsarten recht vielseitig. Das gilt nicht für den Folgebefehl ADDX, mit dem der Übertrag (im Extendbit) aus einer vorhergehenden Addition berücksichtigt wird. ADDX kann nur eingesetzt werden für Additionen mit der Adressierungsart 'Speicher zu Speicher' –(Ax),–(Ay) oder 'Datenregister zu Datenregister' Dx,Dy. Weil in unserem Fall der eine Operand im Speicher steht und der andere in einem Datenregister, laden wir den Speicheroperanden in ein zweites Datenregister D0, führen die Addition aus und speichern das Ergebnis wieder ab, damit es später bei MAIN4 angezeigt werden kann. An diesem einfachen Beispiel wird bereits deutlich, daß die möglichen Adressierungs-

arten den Gang der Programmierung mitbestimmen. Zwar gibt es fast immer mehrere Lösungsmöglichkeiten für eine gestellte Aufgabe, den am schnellsten ausführenden Code erhält man jedoch nur, wenn man möglichst viele Bearbeitungen in Datenregistern ausführen kann und wenn man mit wenigen MOVE-Befehlen auskommt.

Der Programmteil **SUBIT** ist wie ADDIT aufgebaut.

Multiplikation. Der 68000 hat zwei binäre Multiplikationsbefehle, nämlich MULU und MULS für vorzeichenfreie und vorzeichenbehaftete Zahlen (Vorzeichen im vordersten Bit). Die beiden Faktoren der Multiplikation haben eine automatische Verarbeitungsbreite von jeweils 16 Bit (Wortbreite), das Ergebnis eine Breite von 32 Bit. Es muß immer ein Datenregister beteiligt sein; vor der Multiplikation enthält es den Faktor 2, danach das Ergebnis als Langwort. Der Quelloperand (1. Faktor) kann dagegen sehr vielseitig adressiert sein.

Bei **MULIT** sollen entgegen diesen Einschränkungen zwei Langworte miteinander multipliziert werden, so daß sich ein Ergebnis von 64 Bit ergibt. Die Programmzeilen zwischen MULIT und NXTFUNC2 zeigen, daß dafür bereits ein etwas größerer Aufwand getrieben werden muß. Dabei enthalten die Zeilen vor MULIT1 die Vorbereitungen. Ab diesem Label erfolgen Teil-Multiplikationen und ab MULIT3 werden die Teilergebnisse zusammengetragen. Dazu im einzelnen:

Der für den 68000 gewählte Lösungsweg entspricht einer Multiplikation auf dem Papier. Bei einer Rechenaufgabe von z. B. 55 * 23 multiplizieren wir den ersten Faktor 55 ziffernweise mit dem zweiten, schreiben die Teilergebnisse stellenversetzt und addieren sie dann zum Gesamtergebnis:

```
  55 * 23
 ---------
 110
  165
 ---------
 1265
 =========
```

Bei Multiplikationen mit MULU und MULS werden jeweils 4 Hexziffern des einen Faktors mit 4 des anderen multipliziert. Die Teilergebnisse sind daher um vier Stellen gegeneinander verschoben zu schreiben. Wir zeigen das jetzt an der hexadezimalen Rechenaufgabe 223333 * 445555:

```
      22 3333 * 44 5555 (hex)
  ------------------------------
    1110 EEEF                    (3333 * 5555, Low1 * Low2)
  B 554A                         (  22 * 5555, High1 * Low2)
  D 998C                         (  44 * 3333, High2 * Low1)
908                              (  44 *   22, High2 * High1)
  ------------------------------
920 FFE6 EEEF (hex)
  ==============================
```

Aus Gründen der Bequemlichkeit legen wir in MULIT Kopien des 2. Faktors (in D3) in den Datenregistern D2, D0, vom ersten Faktor in D1 an. Die erste Multiplikation der niederwertigsten Teile erfolgt bei MULIT1, und zwar 'Speicher gegen Datenregister', um diese Adressierungsart auch einmal zu zeigen. Das Ergebnis im unteren Wort von D3 wird sofort nach OPER1H+6 in den Speicher gebracht. Warum folgen nun die Befehle CLR.W D3 und SWAP D3? CLR.W D3 bringt die untere Hälfte des Datenregisters auf 00. SWAP D3 vertauscht die beiden Hälften, so daß im vorderen Wort 00 steht und im zweiten Wort das höherwertige Zwischenergebnis der ersten Multiplikation. Mit diesem Trick haben wir den Versatz um 4 Stellen für die spätere Addition der weiteren Zwischenergebnisse geschafft.

Ab MULIT2 folgen weitere drei Teil-Multiplikationen mit 16 x 16 Bit. Der Befehl SWAP wird dabei benutzt, um das vordere Wort der Faktoren für Multiplikationen zu aktivieren. Die Kommentare im Programm zeigen, was jeweils womit multipliziert wird. Man muß sich etwas in diese Zeilen und die nachfolgende Addition der Teilergebnisse für das Verständnis vertiefen.

Division. Auch die Division mit den 68000er-Befehlen DIVU und DIVS (ohne und mit Vorzeichen) hat Besonderheiten: Die Abarbeitung erfolgt jeweils in der Form Dividend / Divisor = 32 / 16 Bits. Ein Datenregister muß beteiligt sein und vor der

Ausführung den Dividenden enthalten. Nach der Ausführung enthält dieses Register im vorderen Wort den Divisionsrest und im unteren Wort den Quotienten. Der Divisor kann in einem anderen Datenregister oder im Speicher stehen. Und es gibt weitere Hinweise: Division durch 0 führt zu einer Ausnahmesituation (Exception); wenn das Ergebnis größer als 16 Bit sein würde (größer $FFFF), dann führt der Befehl nicht aus und das Datenregister mit dem Dividenden wird nicht zerstört. Ein Beispiel dafür: $80000 / 2 führt nicht aus. In diesem Fall wird das V-Bit im Status gesetzt, und man kann die Division dann in zwei Schritte aufteilen, um den höherwertigen und den niederwertigen Teil des Quotienten zu erhalten. Diese Entscheidung erfolgt in der Programm-Liste beim Label DIVIT1.

Die in der Liste enthaltenen Anregungen zur Implementierung der vier binären Grundrechenarten zeigen, daß man mit wenig Aufwand zu Berechnungen mit einer erhöhten Genauigkeit kommt. Bei Additionen, Subtraktionen und Verschiebungen hat man die Rolle des Extendbits zu beachten. – Etwas schwieriger wird es, wenn man Multiplikation und Division mit noch größeren Werten ausführen will. In jedem Falle ist genau zu überlegen, welche Adressierungsart für die Operanden einen knappen und trotzdem schnell ausführenden Code erzeugt.

ADDI 1

```
* ADDI1
* Ausfuehrung binaerer Rechenarten, interaktiv

cr        equ       $0d            Carriage Return
lf        equ       $0a            Linefeed
escape    equ       $1b            Escape-Taste
space     equ       $20            Zwischenraum

 SECTION  ONE

main      movem.l   a3-a5,regsave  Register sichern
mainl     jsr       crlf           Neue Zeile
          move.l    #text1,d0      Ueberschrift anzeigen
          jsr       printline
          jsr       crlf
          move.l    #text2,d0      Zahlen anfordern
          jsr       printline
          lea.l     inbuff,a5      Zeiger auf Eingabepuffer
          move.b    #0,(a5)        Delimiter vorne in den Puffer
          jsr       getline        1 Zeile holen
          cmp.b     #escape,d0     Will man abbrechen?
          beq       abbruch        ja
          jsr       crlf           Neue Zeile

mainla    lea.l     inbuff,a5      Zeige auf Zeilenbeginn
          clr.l     operlh+4       1. Operand-low=0
          clr.l     operlh         dito high
          clr.b     cflag          Steuerung der Umwandlung
main2     move.b    (a5)+,d0       1. Zeichen holen
          beq.s     mainl          Zeile ist leer
          cmpi.b    #space,d0      Fuehrende Leerzeichen?
          beq.s     main2          ja, uebergehen
          jsr       conv           Ziffern filtern, Eingabe umwandeln
          bcs       ueberl         Ueberlauf beim Shiften
          clr.l     d2             Empfangsregister vorbereiten
          clr.l     d3             dito
          cmpi.b    #space,d0      Folgt Space als Trenner?
          bne       main4          Abbruch nach 1. Operanden
          move.b    (a5)+,opart    Hier muss Operator folgen, merken
          move.b    (a5)+,d0       Space erwartet
          cmpi.b    #space,d0
          bne       main4          nein, Eingabefehler
          not.b     cflag          Steuerung: Umwandlung in Register
          move.b    (a5)+,d0       Es muss Ziffer folgen
main2a    jsr       conv           2. Operanden filtern/zu binaer umwandeln
          bcs       ueberl         2. Operand zu gross
          move      #0,ccr         Extendbit im Status loeschen
          cmpi.b    #'+',opart     Welche Operation ist beabsichtigt?
          bne.s     nxtfunc        andere als Addition

addit     add.l     d3,operlh+4    niederwertiges Langwort addieren
          move.l    operlh,d0      wegen moegleicher Adressierungen b. addx
          addx.l    d2,d0          und hoeherwertiges, in den Speicher
          move.l    d0,operlh      Ergebnis abspeichern
          bra       main4          jetzt ausgeben

nxtfunc   cmp.b     #'-',opart     Subtraktion?
          bne.s     nxtfuncl       nein
subit     sub.l     d3,operlh+4    niederwertigen Teil abziehen
          move.l    operlh,d0      hoeherweriger Teil Minuend
          subx.l    d2,d0          minus Subtrahend = Differenz
          move.l    d0,operlh      Ergebnis fuer Ausgabe
          bra       main4          Zur Ausgabe
```

Fortsetzung

ADDI 1 (Fortsetzung)

```
nxtfunc1  cmp.b     #'*',opart      Multiplikation?
          bne       nxtfunc2        nein

* Multiplikation bis 32x32 Bit, sonst Abbruch
* Die einzelnen Schritte verarbeiten 16x16 Bit vorzeichenfrei
mulit     tst.l     operlh          Teil High muss 0 sein
          bne       ueberl
          tst.l     d2              ebenso D2
          bne       ueberl
          move.l    d3,d2           sichere, kopiere 2. Operanden nach D2
          move.l    d3,d0           dito nach D0
          move.l    operlh+4,d1     Faktor 1 holen
mulit1    mulu      operlh+6,d3     Low1*Low2 nach D3
          move.w    d3,operlh+6     und in niederstes Wort abspeichern
          clr.w     d3              untern Teil loeschen
          swap      d3              Registerhaelften vertauschen f. ADD
          swap      d2              High2 fuer MULU aktiv machen
mulit2    mulu      d1,d2           Low1*High2 nach D2
          swap      d1              High1 aktivieren f. MULU
          mulu      d0,d1           = Low2*High1 nach D1
          swap      d0              High2 aktivieren durch Tausch
          mulu      operlh+4,d0     = High1*High2 nach D2

* Zwischenwerte jetzt: D0=hoechstwertiger Teil, D1/D2/D3 mittlerer Teil
* und niederwertiger in operlh+6 im Speicher, nun aufaddieren
mulit3    add.l     d1,d2
          addx.l    d2,d3           mittlere Werte jetzt in D3
          move.w    d3,operlh+4     unteren Teil der Mitte abspeichern
          clr.w     d3              loeschen und umstellen
          swap      d3              Vorderes Wort nun 00
          add.l     d3,d0           Hoechstwertiger Teil, weiter versetzt
          move.l    d0,operlh       Hoechstwertige 16 Bits
          bra       main4           Zur Ausgabe
nxtfunc2  cmp.b     #'/',opart      Division?
          bne       main1
divit     tst.l     d2              Eingabe zu gross?
          bne       ueberl          ja
          tst.l     operlh          und hier?
          bne       ueberl          auch
          tst.w     operlh+4        Divisor darf nur 16 Bits haben
          bne       ueberl          zu gross!
          move.l    operlh+4,d0     Dividend in das Datenregister
          clr.w     operlh+4        Ergebnis teilw. ausnullen
          divu      d3,d0           D0/D3 ausfuehren
divit1    bvc       divitz          Kein Ergebnisueberlauf, gut
          move.l    d0,d2           Kopie des Dividenden anfertigen
          clr.w     d2              Hoeheren Teil aktivieren
          swap      d2              Registerhaelften tauschen
          divu      d3,d2           Hoeherwertiges Ergebnis in D2.W
          move.w    d2,operlh+4     abspeichern
          clr.w     d2              Teil-Quotienten loeschen, Rest erhalten
          andi.l    #$FFFF,d0       Nur Wort Low des Dividenden erhalten
          or.l      d2,d0           Reste zusammenkopieren
          divu      d3,d0           D0/D3, niederwertiger Teil

divitz    move.w    d0,operlh+6     Niederwertiges Ergebnis abspeichern

main4     lea.l     operlh,a5       Ergebnisausgabe
          jsr       longout
          jsr       longout
          bra       main1

abbruch   movem.l   regsave,a3-a5   Register zurueckholen
          jmp       term            Programm beenden
```

Fortsetzung

ADDI 1 (Fortsetzung)

```
ueberl    move.l    #text3,d0       Fehlermeldung
          jsr       printline
          bra       main1

* conv wandelt string zu binaer
conv      jsr       hexfilt         Erlaubte Eingabe?
          bmi.s     conv3           nicht (mehr) hex
          cmpi.b    #$41,d0         Ziffern A...F/a...f berichtigen
          bcs.s     conv1           war 0...9
          addi.b    #9,d0
conv1     andi.w    #$000f,d0       maskiere vordere Bits
          move.w    #3,d1           Schleifenzaehler einrichten
          tst.b     cflag           Steuerung der Umwandlung
          bmi.s     conv4           Benutze Datenregister
* conv2 = Eingabe in das Memory unwandeln
conv2     asl.w     operlh+6        Wort low links schieben
          roxl.w    operlh+4        Das geht bei Speicheroperanden
          roxl.w    operlh+2        nur wortweise
          roxl.w    operlh          links schieben mit Extendbit fertig
          dbcs      d1,conv2        Schleife 4x durchlaufen
          bcs.s     conv3           Ueberlauf
          or.b      d0,operlh+7     geholten Wert hinzu-odern
conv2a    move.b    (a5)+,d0        Naechstes Zeichen holen
          bne.s     conv
conv3     rts

* conv4 = Eingabe in die Datenregister d3,d2 umwandeln
conv4     asl.l     #1,d3           Niederwertiger Teil
          roxl.l    #1,d2           Hoeherwertiger mit Extend-Bit
          dbcs      d1,conv4        Schleife 4x durchlaufen
          bcs.s     conv3           Bei Ueberlauf
          or.b      d0,d3           Eingabeziffer hinzugeben
          bra.s     conv2a          Folgezeichen holen

* LONGOUT, WORDOUT und BYTEOUT
* bringen den Inhalt beliebiger Langwoerter, Woerter, Bytes
* zur Bildschirmausgabe. Der Adresszeiger muss im Register A5
* vor dem Aufruf uebergeben worden sein.
* Es wird die Routine HEXOUT des Abschnittes Memory-Dump benutzt
longout   bsr.s     byteout         Vorderstes Byte ausgeben
          bsr.s     byteout         2. Byte des Langwortes
wordout   bsr       byteout         1. Byte des Wortes
byteout   move.b    (a5)+,d0        Ein Byte ausgeben
          jmp       hexout          RTS bei HEXOUT

* Textkonstante
text1     dc.b      'Binaere Addition, ESC fuehrt zu TOS zurueck ',0
text2     dc.b      'Bitte 2 Hexzahlen und Operator eingeben :  ',0
text3     dc.b      'Zahl zu gross! ',0

* Arbeitsspeicher
regsave   ds.l      3               Ablage von Registern
operlh    ds.l      2               Feld des 1. Operanden
inbuff    ds.b      80              Allg. Eingabepuffer
filtflag  ds.w      1               Flag-Variable
cflag     ds.b      1               Steuerung der Umwandlung
opart     ds.b      1               Beabsichtigte Operation

 INCLUDE  GEMDOSFN
 END
```

Kapitel 43

Dezimales Rechnen

Gegenüber den binären Rechenarten sind die dezimalen auf dem 68000 nur schwach unterstützt: Die Befehle **ABCD**, **SBCD** und **NBCD** bewirken Addition, Subtraktion und Bildung eines 9-er oder 10-er Komplements. Sie arbeiten pro Befehlsschleife jeweils nur an 1 Byte, d. h. an 2 Ziffern in der Darstellungsform BCD (Binary Coded Digit). In der binären Darstellung der Ziffern sind also nur 0000 ... 1001 zugelassen. Die weiteren Beschränkungen betreffen die zulässigen Adressierungsarten. Die Befehle arbeiten entweder nur 'Datenregister zu Datenregister' oder 'Speicher zu Speicher'. Der Programmteil **ADDSUB** enthält Verschiebebefehle um X Stellen in Datenregistern und die Bewahrung des Extendbits (Übertrag) im Bedingungscode-Register (CCR des Status).

Überblick. In der nachfolgenden Liste eines Programmes sollen lediglich die Programmierungsmöglichkeiten aufgezeigt werden. Es wurde daher darauf verzichtet, eine Art von Tischrechner für den Anwender aufzubauen. Stattdessen wird hier mit Operanden gearbeitet, die der Assembler als Konstante im Speicher abgelegt hat. Der Leser wird bei ihrem Studium erkennen, daß die dezimalen Befehle unter Benutzung von Datenregistern eigentlich zu vergessen sind, weil umständlich. Als abschreckendes Beispiel mögen die Zeilen zwischen den Labeln ADDIT und MAIN4 dienen. Sie rufen zweimal (für erhöhte Genauigkeit) das Unterprogramm ADDSUB bzw. ADDSUB1 auf. Bei den Abschnitten SUBDEZ, ZEHNER und NEUNER werden jedoch die Adres-

sierungsarten 'Speicher zu Speicher' mit –(Ax),–(Ay) benutzt. Hier zeigt sich, daß man dezimale Additionen und Subtraktionen mit beliebiger Genauigkeit elegant ausführen kann, und zwar mit 2 Befehlszeilen. Beim dezimalen Rechnen gilt also nicht notwendig, daß man unter Benutzung von Datenregistern ein schneller ausführendes Programm erhält.

Zum Programm. Die nachstehend gelisteten Routinen arbeiten mit einer Genauigkeit von 64 Bit, d. h. mit 16 signifikanten Dezimalziffern in der Eingabe und im ERGEBNIS. Ab Label ADDIT wird die umständliche Addition in Datenregistern vorgeführt. Wenn man sie benutzt, geschieht die dezimale Bearbeitung byte-weise, anders als in den binären Rechenarten, wo man Addition und Subtraktion mit einem Befehl auf ein Langwort ausüben kann. Die Datenregister werden mit den Operanden geladen, dann erfolgt JSR ADDSUB bzw. JSR ADDSUB1 für den höherwertigen Teil. Nach den Vorbereitungen nun ein Blick auf das Unterprogramm ADDSUB:

Bei der dezimalen Addition gibt es keinen Befehl vom Stile "addiere ohne Übertrag". Wir müssen daher mit einem definierten Status "ohne Übertrag" beginnen, den wir in die Variable STATUS bringen. Nach jeder Addition wird der erreichte Status dort abgespeichert (MOVE.W SR,STATUS) und jeweils bei Schleifenbeginn von dort neu geladen (MOVE.W STATUS,CCR). Dabei ist kurz zu erinnern, daß der Befehl 'MOVE to SR' für den Supervisor-Modus privilegiert ist. Dieses Merken des erreichten Status ist notwendig, weil die rechts verschiebenden Befehle LSR.L sowohl das Carry- wie auch das Extendbit verändern. In den Datenregistern D1 und D2 wird immer um die Position eines Bytes, also um 8 Bit, in das Additionsfenster geschoben. Das Verfahren ist also umständlich.

DEZADD

```
* DEZADD
* Ausfuehrung dezimaler Rechenarten
* Addition, Subtraktion, Komplementbildung

 SECTION  ONE

main     movem.l   a3-a5,regsave  Register sichern
mainl    jsr       crlf           Neue Zeile
         move.l    #text1,d0      Ueberschrift anzeigen
         jsr       printline
         jsr       crlf
* Ausfuehrung der Addition Datenregister zu Datenregister
* Genauigkeit 64 Bit = 16 Dezimalziffern
addit    move.l    #text2,d0      Anzeige Addition
         jsr       printline
         move.l    oper11,d1      1. Operand Low
         move.l    oper21,d2      2. Operand Low geladen
         lea       ergebnis+8,a5  Bereite Ablage ,-(a5) vor
         jsr       addsub         erste 32 Bits dezimal addieren
* Jetzt die hoeherwertigen 8 Bytes addieren, 2. Durchgang
         move.l    oper1h,d1      1. Operand High
         move.l    oper2h,d2      2. Operand High
         jsr       addsub1        weiter addieren mit letztem Status
main4    jsr       ergaus         Ergebnis anzeigen
         move.w    status,ccr     X-Bit im letzten Status pruefen
         bcs       ueber1         Ueberlauf anzeigen
main5    jsr       crlf           Neue Zeile
         move.l    #text3,d0      Nun Subtraktion anzeigen
         jsr       printline
* Vorbereitung der dezimalen Subtraktion OPER1 - OPER2
         lea.l     oper1h+8,a3    Zeige hinter ersten Operanden
         lea.l     oper2h+8,a4    Dito hinter den 2. fuer ,-(A4)
         move.w    #15,d1         Register als Zaehler
         move      #$04,ccr       Fange ohne Uebertrag an
subdez   sbcd      -(a4),-(a3)    Bytes in (A3) - (A4)
         dbf       d1,subdez      Bleibe in der Schleife
         move.l    oper1h,ergebnis    Ergebnis uebertragen
         move.l    oper11,ergebnis+4  Speicher zu Speicher
         jsr       ergaus         und anzeigen
         jsr       crlf
* Nun werden das Zehner- und das Neuner-Komplement dezimal gebildet
         move.l    #text4,d0      Anzeige Zehnerkomplement
         jsr       printline
         jsr       crlf
         lea.l     ergebnis+8,a4  hinter altes Ergebnis setzen
         move.w    #15,d1         Register als Zaehler
         move      #$04,ccr       Fange  ohne  Uebertrag an
zehner   nbcd      -(a4)          Bytes in (A4),10-er Komplement, Schleife
         dbf       d1,zehner      Bleibe in der Schleife
         jsr       ergaus         und anzeigen
         jsr       crlf
*
         move.l    #text6,d0      Anzeige Neunerkomplement vorbereiten
         jsr       printline
         jsr       crlf
         lea.l     ergebnis+8,a4  hinter altes Ergebnis setzen
         move.w    #15,d1         Register als Zaehler
         move      #$14,ccr       Fange  mit  Uebertrag an
neuner   nbcd      -(a4)          Neunerkomplement bilden, Scleife
         dbf       d1,neuner      Bleibe in der Schleife
         jsr       ergaus         und anzeigen
         jsr       crlf
         jsr       conin          Tastatur abfragen, warten
```

Fortsetzung

DEZADD (Fortsetzung)

```
          cmpi.b    #$1b,d0          Escape gedrueckt?
          beq.s     abbruch          ja
          bra       main1

abbruch   movem.l   regsave,a3-a5    Register zurueckholen
          jmp       term             Programm beenden
ueber1    move.l    #text5,d0        Fehlermeldung, Ueberlauf/Borgen
          jsr       printline        in hoechster Stelle
          bra       main5

* Unterprogramm der Addition byteweise in Datenregistern
addsub    move.w    #$04,status      X-Bit=0, Z-Bit=1 im Status vorbesetzen
addsub1   move.w    #3,d3            3x rechts schieben vormerken f. DBF
addloop   move.w    status,ccr       Status fortsetzen
          abcd      d1,d2            Addition, Ergebnis in D2
          move.w    sr,status        Condition Code merken, spez. X-Bit
          move.b    d2,-(a5)         Ergebnisbyte in den Speicher
          lsr.l     #8,d1            Naechstes Byte ins ABCD-Fenster
          lsr.l     #8,d2            Dito 2. Operand, 8 Bits schieben
          dbf       d3,addloop       Dekrementiere D3, Rueckverzweigung
          rts
ergaus    lea.l     ergebnis,a5      Ergebnisausgabe
          jsr       longout          a.a.O. dokumentiertes Programm
          jsr       spaceout         Space nach 8 Zeichen ausgeben
          jsr       longout
          rts

spaceout  move.b #$20,d0
          jmp conout

* Textkonstante
text1     dc.b      'Dezimales Rechnen, ESC fuehrt zu TOS zurueck ',0
text2     dc.b      'Addition ABCD: ',0
text3     dc.b      'Subtraktion SBCD: ',0
text4     dc.b      'Negation des Ergebnisses, NBCD, 10er-Komplement: ',0
text5     dc.b      '  Ueber-/Unterlauf ',0
text6     dc.b      '9-er Komplement des Ergebnisses: ',0

* Konstante dezimale Rechenwerte
oper1h    dc.l      $56000111        Codierung als Dezimalzahl im Speicher
oper1l    dc.l      $00000099
oper2h    dc.l      $35000000
oper2l    dc.l      $00000002
* Arbeitsspeicher
regsave   ds.l      3                Ablage von Registern
ergebnis  ds.l      3                Ergebnis- und Ausgabefeld
status    ds.w      1                Merke das CCR-Register

 INCLUDE  GEMDOSFN
 END


* Arbeitsspeicher
regsave   ds.l      3                Ablage von Registern
oper1h    ds.l      2                Feld des 1. Operanden
inbuff    ds.b      80               Allg. Eingabepuffer
filtflag  ds.w      1                Flag-Variable
cflag     ds.b      1                Steuerung der Umwandlung
opart     ds.b      1                Beabsichtigte Operation

 INCLUDE  GEMDOSFN
 END
```

Kapitel 44

Zahlenwandlung

Die Befehle für binäres Rechnen sind auf dem 68000 vielseitiger eingerichtet als diejenigen für dezimales Rechnen. Man wird sich daher in vielen Fällen rechnerintern für die binäre Darstellung und Verarbeitung von Zahlen entschließen und lediglich bei der Ein- und Ausgabe mit Dezimalzahlen arbeiten. Damit entsteht das Problem der Zahlenumwandlung dezimal zu binär bei der Eingabe und umgekehrt bei der Ausgabe. Der laufende Abschnitt enthält ein kleines interaktives Programm, das diese immer wieder benötigten Umwandlungen besorgt.

Einführung. Wie schon in anderen Beispielen wird bei der Eingabe von Zahlen das Unterprogramm GETLINE benutzt. Nach dem Zeilenabschluß mit CR wird die erhaltene Zeile zeichenweise auf die allein zulässigen Ziffern 0 ... 9 gefiltert. Für die Aufnahme des binären Ergebnisses aus dem Eingabestrom, der von links nach rechts durchgelesen wird, sind die Datenregister D2 und D3 vorgesehen, weil die später notwendigen Multiplikationen nur unter Beteiligung eines Datenregisters ausgeführt werden können. Dabei ist wie folgt disponiert: D3 wird als Langwortregister benutzt, D2 jedoch nur als Wortregister. Die größte darstellbare Binärzahl hat damit 48 Bits; das entspricht dezimal knapp einer 3 mit 14 Nullen dahinter, also einer sehr großen Dezimalzahl. Das hier nicht benutzte höherwertige Langwort von D2 könnte man in besonderen Anwendungen noch für die Aufnahme von Vorzeichen oder Exponenten benutzen, wenn man mit Gleitkom-

mazahlen arbeiten will. Alternativ kann man natürlich auch den zulässigen Wertebereich erhöhen.

Ablauf. Während der Zahleneingabe ist mit jeder neu angetroffenen, gültigen Dezimalziffer das bisherige Zwischenergebnis der Umwandlung zunächst mit 10 zu multiplizieren, ehe die neue Ziffer im Register D0 hinzuaddiert wird. Das geschieht im Unterprogramm DEZHEX. Das Register D1 enthält dort den konstanten Faktor 10. Da auf dem 68000 Multiplikationen immer nur in Portionen von 16 x 16 Bit ausgeführt werden, wobei ein Ergebnis von 32 Bit in einem Datenregister entsteht, sind drei Teil-Multiplikationen auszuführen: Der höchstwertige Teil des Zwischenergebnisses steht in D2.W, der mittelwertige "links" im Register D3 und der niederstwertige in D3.W. Das Ergebnis der Multiplikationen ist anschließend stellenrichtig aufzuaddieren. Dabei haben die Befehle CLR.W Datenregister und SWAP einige Bedeutung, um die Zwischenergebnisse entsprechend ihrer Wertigkeit auszurichten. Ab Label DEZHEX1 wird die letzte empfangene Ziffer zum bisherigen Ergebnis hinzuaddiert. Nach Abspeicherung der Datenregister kann später bei MAIN4 das Umwandlungsergebnis in hexadezimaler Darstellung ausgegeben werden.

Ab Label HORNER wird die eben erzeugte Binärzahl in die Darstellungsform BCD zurückverwandelt, und zwar im Empfangsfeld OPER2. Aus dieser Codierung gelangt man einfach wieder zur druckbaren Codierung in ASCII, wenn man zu jedem Halbbyte eine hexadezimale 30 hinzuaddiert. Unsere frühere Routine LONGOUT besorgt das später automatisch.

Der Algorithmus der Rückumwandlung beruht auf dem Horner-Schema. Die binären Ziffern des Operanden in D2/D3 werden bei HORNLOOP einzeln nach links in das Extendbit herausgeschoben. Bei HORNADD erfolgt mit ABCD –(A5),–(A4) eine Verdoppelung des bisherigen Zwischenergebnisses in der Darstellungsform BCD, indem es auf sich selbst aufaddiert wird, und zwar unter Einbeziehung des durch die vorgenannte Linksverschiebung gewonnenen Extendbits. Der Lösungsweg ist damit analog zum Unterprogramm DEZHEX. Dort ist mit jedem Eintref-

fen einer neuen Ziffer das bereits vorhandene Zwischenergebis mit 10 zu multiplizieren. Bei Horner hat jedes Bit die Stellenwertigkeit 2; daher die Verdoppelung des Zwischenergebnisses.

Beschleunigung der Umwandlung. Der Umwandlungsprozeß kann nach folgenden Überlegungen beschleunigt werden: Wenn im Datenregister D2.W irgendwelche Bits zu 1 gesetzt sind, dann müssen wir alle 48 Bits des binären Operanden in D2/D3 in die Umwandlung einbeziehen. Wenn nicht, dann können wir bei HORN2 und HORN3 prüfen, ob wir mit noch weniger Verschiebungen auskommen. In diesem Fall sind die Hälften des Registers D3 stellenrichtig in das Verschiebefenster vorne in D3 zu transportieren. Mit einigen Programmzeilen mehr kann man die Vorgänge insgesamt noch beschleunigen, was bei ausgabeintensiven Jobs lohnen mag, denn eine Zahlenwandlung ist immer etwas zeitaufwendig.

WANDLUNG

```
* WANDLUNG
* Umwandlung dezimal zu binaer und umgekehrt, interaktiv

cr        equ       $0d            Carriage Return
lf        equ       $0a            Linefeed
escape    equ       $1b            Escape-Taste
space     equ       $20            Zwischenraum

 SECTION  ONE

main      movem.l   a3-a5,regsave  Register sichern
mainl     jsr       crlf           Neue Zeile
          move.l    #text1,d0      Ueberschrift anzeigen
          jsr       printline      und Dezimalzahl anfordern
          jsr       crlf
          lea.l     inbuff,a5      Zeige auf Tastaturpuffer
          move.b    #0,(a5)        Delimiter vorne in den Puffer
          jsr       getline        1 Zeile holen
          cmp.b     #escape,d0     Will man abbrechen?
          beq       abbruch        ja
          jsr       crlf           Neue Zeile
* Dezimale Eingabe nach binaer umwandeln:
mainla    lea.l     inbuff,a5      Zeige auf Zeilenbeginn
          move.w    #10,d1         Zahlenbasis, Faktor konstant 10
          clr.l     d0             fuer spaetere Addition
main2     move.b    (a5)+,d0       1. Zeichen holen
          beq.s     mainl          Zeile ist leer
          cmpi.b    #space,d0      Fuehrende Leerzeichen?
          beq.s     main2          ja, uebergehen
          jsr       dezhex         Ziffern filtern, Eingabe umwandeln
          move.l    d2,oper1       Ergebnis ablegen fuer Anzeige
          move.l    d3,oper1+4     dito Low

* Binaerzahl zurueckverwandeln zu Dezimalzahl
horner    clr.l     oper2          Ergebnisfeld = 00
          clr.l     oper2+4
          tst.w     d2             Sind Ziffern enthalten?
          beq.s     horn2          nein, man kann abkuerzen
horn1     move.w    #47,d1         Zaehler fuer 48 Durchlaufe
          bra.s     hornloop
horn2     move.l    d3,d1          Inhalt pruefen
          beq.s     horn4          Operand ist Null
          swap      d1             Hoehere Haelfte untersuchen
          beq.s     horn3          wir koennen abkuerzen
          move.w    d1,d2          Hoeherwertigen Teil links ins Fenster
          swap      d3             Auch hier nach links ruecken
          move.w    #31,d1         32 Durchlaeufe
          bra.s     hornloop
horn3     tst.w     d3             Haben wir insgesamt eine Null?
          beq.s     horn4          ja, nichts ist zu wandeln
          move.w    #15,d1         Nur 16x Aussenschleife durchlaufen
          move.w    d3,d2          16 Bits nach vorne bringen
hornloop  asl.l     #1,d3          Teil Low 1x links schieben
          roxl.w    #1,d2          dito High mit Extendbit
* Das herausgeschobene Bit ist jetzt im Extendbit
          moveq     #7,d4          Zaehler fuer 8 Byte-Additionen
          lea.l     oper2+8,a4
          lea.l     oper2+8,a5
hornadd   abcd      -(a5),-(a4)    Ergebnisfeld verdoppeln
          dbf       d4,hornadd     8 Bytes addieren
          dbf       d1,hornloop    Aussenschleife 24x durchlaufen
horn4     nop                      Dummy, rts, wenn Unterprogramm
```

Fortsetzung

WANDLUNG (Fortsetzung)

```
main4     lea.l     oper1,a5        Ergebnisausgabe
          jsr       longout
          jsr       longout
          jsr crlf
          jsr longout
          jsr longout
          bra       main1

abbruch   movem.l   regsave,a3-a5   Register zurueckholen
          jmp       term            Programm beenden

* dezhex wandelt string zu binaer in den Registern d2, d3
dezhex    clr.l     d2              Ergebnisfeld = 0, hoeherwertiger Teil
          clr.l     d3              dito, niederwertiger Teil
dezhex0   jsr       dezzif          Erlaubte Eingabe?
          bmi.s     dezhex3         nicht (mehr) hex
          move.l    d3,d4           Kopie machen + hoeherwertigen Teil
          clr.w     d4              als Faktor isolieren
          swap      d4              erledigt, jetzt bisheriges Ergebnis
          mulu      d1,d2           mit Zahlenbasis multiplizieren
          mulu      d1,d4           Mittelfeld
          mulu      d1,d3           hoeherwertiger Teil
          move.w    d3,d5           Kopie Zwischenergebnis sichern
          clr.w     d3              Mittelfeld loeschen
          swap      d3              austauschen
          move.w    #0,ccr          Extendbit loeschen
          add.w     d4,d3           Mittelfeld aufaddieren
          swap      d3              Mittelfeld jetzt richtig vorne in D3
          move.w    d5,d3           Niederwertigen Teil einkopieren
          clr.w     d4              Mittelfeld loeschen
          swap      d4              stellenrichtig f. Addition machen
          addx.l    d4,d2           Addiere mit Extend, vorderer Teil
          move.w    #0,ccr          Extendbit loeschen
dezhex1   andi.b    #$0f,d0         maskiere vordere Bits, Eingabeziffer
          add.l     d0,d3           Eingabeziffer hinzuaddieren
          bcc.s     dezhex2         Wenn kein Uebertrag entstand
          addq.l    #1,d2           Uebertrag einaddieren
dezhex2   move.b    (a5)+,d0        Naechstes Zeichen holen
          bne.s     dezhex0         und umwandeln
dezhex3   rts

* DEZZIF ist ein Filter fuer Dezimalziffern 0...9
dezzif    cmpi.b    #$3a,d0         Groesser als '9'?
          bcc.s     nothex          ja
          cmpi.b    #'0',d0         Unter '0'?
          bcs.s     notdez          ja, FILTFLAG=negativ
          clr.w     filtflag        FILTFLAG=0
          rts
notdez    move.w    #$ffff,filtflag Flag wird negativ
          rts

* Textkonstante
text1     dc.b      'Zahlenwandlung, ESC fuehrt zu TOS zurueck ',cr,lf
          dc.b      'Bitte eine Dezimalzahl eingeben  ',cr,lf
          dc.b      'Maximal 14 Ziffern:  ',cr,lf
          dc.b      'Dez-Hex und dann Hex-Dez',cr,lf,0

* Arbeitsspeicher
regsave   ds.l      3               Ablage von Registern
oper1     ds.l      2               Feld des 1. Operanden
oper2     ds.l      2               2. Operand
inbuff    ds.b      80              Allg. Eingabepuffer
filtflag  ds.w      1               Flag-Variable
```

Kapitel 45

Arrays

Die Adressierungsart "Adreßregister indirekt mit Index" des 68000 läßt sich hervorragend benutzen, um Datensätze und in ihnen Datenfelder gezielt anzusprechen. Solche Adressierungen werden typisch bei der Bearbeitung von Arrays benutzt. Als praktische Anwendung enthält dieser Abschnitt die Vorstufe einer kleinen interaktiven Datenbank, die ein Array im Speicher bildet. Der Leser wird dabei bemerken, daß der Aufbau einer Datenbankstruktur auch in Assembler einfach möglich ist, dazu bedarf es also nicht unbedingt einer höheren Sprache wie C oder PASCAL.

Ziel dieses Programmbeispieles. Es soll eine Datenbank mit zunächst 100 zulässigen Datensätzen (engl.: Record) eingerichtet werden. Jeder Datensatz hat 3 Felder fester Länge, nämlich für Name (20 Zeichen), Ort (20) und Nummer (4). Wir werden mit der Befehlstaste 'e' in der Lage sein, Eintragungen in beliebige Datensätze zu machen, deren Nummer zunächst abgefragt wird. Die Taste 'a' führt zur Anzeige eines Datensatzes mit Nummer. Der Programmlauf wird mit der Taste ESC abgebrochen. Wiederum wird auf zahlreiche in anderen Kapiteln entwickelte Unterprogramme zurückgegriffen, auf die GEMDOS-Funktionen im "GEMDOS-Toolkit", auf GETLINE in "Ein Editor" und auf CONV in "Binäres Rechnen" u. a. Der Leser mag erkennen, daß man mit den genannten Grundleistungen, die man in eine Programmbibliothek aufnehmen sollte, schon zu sehr leistungsfähigen Programmen fortschreiten kann.

Es wird lediglich die Vorstufe einer Datenbank geboten. Eine Such- und Ersetzungsfunktion wurde noch nicht implementiert. Anregungen dafür sind im Abschnitt "Ein Parser" zu finden. Ebenfalls nicht implementiert sind eine Sortierung oder eine Indizierung. Eine Abspeicherung auf Floppy ist nicht vorgesehen. Anregungen dafür findet man im Abschnitt "Serielle Datenübertragung" bei der Ausgabe auf Floppy Disk.

Die Struktur der Datensätze. Mit den sprachlichen Möglichkeiten des Assemblers ist am Programmanfang bei **STRUCTURE** die Struktur eines Datensatzes definiert worden. Es kommt hier auf die Vorgabe fester Feldlängen an. Zwischen den Datenfeldern ist eine hexadezimale 00 in den Feldern **BEGR1 ... BEGR3** als Begrenzer vorgesehen. Bei der Initialisierung der Datenbank wird die Null später in der NLOOP in diese Begrenzerfelder eingesetzt. Nach dieser Maßnahme kann das GEMDOS-Unterprogramm PRINTLINE benutzt werden, um Datenfelder bis hin zum Begrenzer 00 bequem ausgeben zu können. Bei der Definition der Struktur kam es auf Konstante an, die man in der späteren Differenzbildung als einen Offset zur gezielten Ansprache von Datenfeldern innerhalb eines Datensatzes benutzen kann. – Die Konstante **RECSIZE** nimmt die pro Datensatz benötigte Menge von Bytes auf. Der von der Datenbank insgesamt benötigte Speicherplatz steht in **BANKSIZE**. Diese Konstante ergibt sich aus (Satzlänge *Zahl der vorgesehenen Datensätze).

Initialisierung der Datenbank. Die Datenbank muß initialisiert werden, denn es könnte ja jemand auf den Gedanken kommen, sich einen Datensatz anzeigen zu lassen, der noch gar nicht angelegt worden ist. In diesem Fall besteht das Risiko, daß wilde Muster auf dem Bildschirm erscheinen. Ab Label **MLOOP** wird die gesamte Datenbank daher zunächst mit Zwischenraumzeichen (Spaces) gefüllt. Dieser Job ist in 5 Programmzeilen elegant erledigt. Es ginge um eine Zeile kürzer, wenn man schriebe "MOVE.B #SPACE,-(A3)". Der Unterschied kann lediglich in benötigten Taktzyklen liegen. Mit den Spaces sind wilde Anzeigen unter PRINTLINE unterbunden. Nun sind für dieses Unterprogramm Feldbegrenzer zu setzen.

Das Einfüllen von Nullen für PRINTLINE in die Begrenzerfelder BEGR1 ... BEGR3 geschieht nach folgender Grundidee: Mit LEA.L ARRAY,A4 wird das Adreßregister A4 auf den Anfang der Datenbank gesetzt. Einige Zeilen später wird es schrittweise auf den Anfang des folgenden Datensatzes erhöht, und zwar mit ADDA.L #RECSIZE,A4. Es wird also jeweils die Länge eines Datensatzes auf das Adreßregister A4 aufaddiert. Das läuft in der **NLOOP**, deren Schleifenzählung von Datenregister D1 mit einer Vorladung von 100 (MAXZAHL) kontrolliert wird (DBF D1,NLOOP, dekrementiere und rückverzweige immer, bis D1 negativ wird). Innerhalb der NLOOP sind nun Nullen in die Begrenzerfelder einzufüllen. Diese haben einen festen Abstand (Offset) relativ zum Anfang eines Datensatzes. Vor der NLOOP wurden daher die Datenregister D2 ... D4 mit dem Direktwert dieser Abstände vorgeladen. Nun wird die Adressierung "Adreßregister indirekt mit Index" innerhalb der NLOOP verständlich: Register A4 zeigt jeweils auf den Anfang eines Datensatzes. CLR.B (A4,Dx.W), alternativ auch einmal MOVE.B D0,(A4,Dx.W), positionieren mit dem Indexregister Dx relativ zum Anfang eines Satzes und erzeugen eine Null an der effektiven Adresse. So elegant bewegt man sich mit dem 68000 in Strukturen!

Für die im vorstehenden Absatz geschilderte Indizierung eines Adreßregisters dürfen nicht nur Datenregister benutzt werden, sondern auch andere Adreßregister. Es besteht damit die Möglichkeit, die Felder eines sehr fein unterteilten Datensatzes mit mehreren als Index benutzten Registern der CPU mühelos anzusprechen.

Die übrigen Teilprogramme. Bei **MAIN1** ist nach der Initialisierung die kleine Befehlsebene erreicht. Das Menue bei **TEXT1** wird angezeigt, ein Tastendruck wird eingeholt. Von hier geht es weiter zu **EINGABE** (Datensätze neu anlegen), **ANZEIGE** (Datensätze anzeigen) oder zum **ABBRUCH** (zu TOS zurück). Die in den ersten beiden Teilprogrammen benutzten Adressierungsarten benutzen oft wieder "Adreßregister indirekt mit Index", aber auch zur Abwechslung "Adreßregister indirekt mit genanntem Offset". Das führt zu einer Schreibweise wie z. B. 4(A5): Positio-

niere auf eine Adresse, die im Adreßregister A5 enthalten ist, plus 4. Die 4 ist dabei der genannte Offset.

Das Teilprogramm EINGABE hat drei gleichartige Abfolgen. Zuerst wird die hexadezimale Nummer des gewünschten Datensatzes abgefragt. Für diese nicht ganz publikumsfreundliche Art der Abfrage können wir uns schnell auf das Unterprogramm CONV und die Multiplikation in "Binäres Rechnen" abstützen. RECPOS liefert dabei im Register D1 die Speicheradresse des gewählten Datensatzes. Relativ zu dieser Basis werden mit GETLINE Eingaben geholt und in die Felder eingetragen, so bei EING1, EING3 und EING6. Der Transport erfolgt dabei aus dem Eingabepuffer der Tastatur direkt in das Feld. Weil man Fehlbedienungen nicht ausschließen kann, werden die Eingaben nach GETLINE zwangsweise mit einer 00 an der Stelle abgeschlossen, wo die Länge des Datenfeldes normalerweise erreicht ist. Längere Eingaben werden damit "abgehackt". Andererseits wird PRINTLINE immer nur bis zu der Null ausgeben, die als erste im Feld angetroffen wird. Das Abhacken ist kein Beinbruch in einem Demoprogramm, es muß aber in durchdachten Anwendungen aufgefangen werden (interaktive Meldungen).

Die wenigen Zeilen des in diesem Abschnitt dargestellten Beispielprogrammes mögen dem Leser einerseits zeigen, wie einfach man mit dem 68000 im Atari auch bei komplexeren Strukturen umgehen kann, und ihm andererseits Mut geben, sich nach seinen Bedürfnissen eigene Datenstrukturen aufzubauen, zu speichern und abzufragen.

ARRAY

```
* ARRAY
* Ansprache von Datensaetzen und Feldern in diesen
* Eine kleine Datenbank

* Definition von Verkehrszeichen
cr        equ       $0d            Carriage Return
lf        equ       $0a            Linefeed
escape    equ       $1b            Escape-Taste
space     equ       $20            Zwischenraum

* Definition eines Datensatzes
structur  equ       0              Definition eines Satzes, Basis
name      equ       structur       1. Feld in Satz, 20 Zeichen
begr1     equ       name+20        Platz fuer 00-Begrenzer
ort       equ       begr1+1        2. Feld in Satz, 20 Zeichen
begr2     equ       ort+20         Platz fuer einen 00-Begrenzer
nummer    equ       begr2+1        3. Feld im Satz, 4 Zeichen
begr3     equ       nummer+4       Begrenzer
satzend   equ       begr3+1        Ende des Satzes

recsize   equ       satzend-structur Umfang eines Datensatzes
maxzahl   equ       99             100 Datensaetze sind zugelassen
banksize  equ       100*recsize    Platz von 100 Datensaetzen

 SECTION  ONE

main      movem.l   a3-a5,regsave  Register sichern
* Zuerst ganzes Array mit Spaces fuellen
          lea.l     bankend,a3     Ende Datenspeicher+1
          move.w    #banksize,d1   Menge der Bytes
          move.b    #space,d0      Zwischenraum-Zeichen laden
mloop     move.b    d0,-(a3)       Array mit Spaces fuellen
          dbf       d1,mloop       Schleife, bis D1 negativ wird
          move.w    #maxzahl,d1    Zaehler fuer 100 Durchlauefe
          move.w    #begr1,d2      Offset 1. Begrenzer
          move.w    #begr2,d3      Offset 2. Begrenzer
          move.w    #begr3,d4      Offset 3. Begrenzer
          lea.l     array,a4       Zeige auf Anfang des Arrays
* Adressierung Adressregister indirekt mit Index im Datenregister
* Es werden Nullen als Feldbegrenzer eingesetzt:
nloop     clr.b     (a4,d2.w)      Eine 00 nach BEGR1
          clr.b     (a4,d3.w)      Dito nach BEGR2
          move.b    #0,(a4,d4.w)   Dito nach BEGR3
          adda.l    #recsize,a4    Setze auf naechsten Record
          dbf       d1,nloop       Schleife 100x durchlaufen
main1     jsr       crlf           Neue Zeile
          move.l    #text1,d0      Menue anzeigen
          jsr       printline
          jsr       crlf
          lea.l     inbuff,a5      Zeiger auf Eingabepuffer
          move.b    #0,(a5)        Delimiter vorne in den Puffer
          jsr       getline        1 Zeile holen
          cmp.b     #escape,d0     Will man abbrechen?
          beq       abbruch        ja
          jsr       crlf           Neue Zeile

main1a    lea.l     inbuff,a5      Zeige auf Zeilenbeginn
          clr.l     operlh+4       1. Operand-low=0
          clr.l     operlh         dito high
main2     move.b    (a5)+,d0       1. Zeichen holen
          beq.s     main1          Zeile ist leer
          cmpi.b    #space,d0      Fuehrende Leerzeichen?
          beq.s     main2          ja, uebergehen
```

Fortsetzung

ARRAY (Fortsetzung)

```
        cmpi.b   #'a',d0        Befehl Anzeige?
        beq      anzeige
        cmpi.b   #'e',d0        Befehl Eingabe?
        bne      main1          Nein, Menue wieder anzeigen

eingabe jsr      recpos         Positioniere auf den Datensatz
        move.l   #text4,d0      Namensfeld anzeigen
        jsr      printline
        lea.l    inbuff,a5      Zeige auf Tastaturpuffer
        move.b   #0,(a5)
        jsr      getline        Eingabezeile holen
        jsr      crlf
        lea.l    inbuff,a5      Zeige auf Platz des Delimiters
        clr.b    begrl(a5)      Null als Zwangsabschluss der Zeile
        lea.l    array,a4       Zeige auf Anfang des Arrays
        adda.l   d1,a4          Positioniere auf den Satzanfang
eing1   move.b   (a5)+,(a4)+    Datentransport ins Feld NAME
        bne.s    eing1          Weiter bis zur Null
eing2   addi.w   #ort-name,d1   Offset auf das Ortsfeld richten
        move.l   #text5,d0      Ortsfeld anzeigen
        jsr      printline
        lea.l    inbuff,a5      Zeige auf Tastaturpuffer
        move.b   #0,(a5)
        jsr      getline        Eingabezeile holen
        jsr      crlf
        lea.l    inbuff,a5      Zeige auf Platz des Delimiters
        clr.b    20(a5)         Null als Zwangsabschluss der Zeile
        lea.l    array,a4       Zeige auf Anfang des Arrays
        adda.l   d1,a4          Positioniere auf das Ortsfeld
eing3   move.b   (a5)+,(a4)+    Datentransport ins Feld NAME
        bne.s    eing3          Weiter bis zur Null
eing4   addi.w   #nummer-ort,d1   Erhoehe den Offset auf Feld NUMMER
        move.l   #text6,d0      Nummernfeld anzeigen
        jsr      printline
        lea.l    inbuff,a5      Zeige auf Tastaturpuffer
        move.b   #0,(a5)
        jsr      getline        Eingabezeile holen
        jsr      crlf
        lea.l    inbuff,a5      Zeige auf Platz des Delimiters
        clr.b    4(a5)          Null als Zwangsabschluss der Zeile
        lea.l    array,a4       Zeige auf Anfang des Arrays
        adda.l   d1,a4          Positioniere auf das Nummernfeld
eing6   move.b   (a5)+,(a4)+    Datentransport ins Feld NUMMER
        bne.s    eing6          Weiter bis zur Null
eing7   bra      main1

anzeige jsr      recpos         Satz-Nr. holen
        move.l   #text4,d0      Name anzeigen
        jsr      printline
        move.l   #array,d0      Inhalt Namensfeld anzeigen
        add.l    d1,d0          Zeige auf Datensatz mit Nr.
        jsr      printline
        jsr      crlf
        move.l   #text5,d0      Ort anzeigen
        jsr      printline
        move.l   #array,d0      Inhalt Ortsfeld anzeigen
        add.l    d1,d0          Zeige auf Datensatz mit Nr.
        add.l    #ort-name,d0   +Offset auf Ort
        jsr      printline
        jsr      crlf
        move.l   #text6,d0      Nummer anzeigen
        jsr      printline
```

Fortsetzung

ARRAY (Fortsetzung)

```
        move.l    #array,d0        Inhalt Nummernfeld anzeigen
        add.l     d1,d0            Zeige auf Datensatz mit Nr.
        add.l     #nummer-name,d0      +Offset Feld Nummer
        jsr       printline
        jsr       crlf
        bra       mainl
abbruch movem.l   regsave,a3-a5    Register zurueckholen
        jmp       term             Programm beenden

recpos  jsr       numhol           Satz-Nr. holen
        clr.l     d1
        move.w    operlh+6,d1      und uebertragen
        mulu      #recsize,d1      Satzgroesse*Nummer = relative Adresse
        rts

numhol  move.l    #text2,d0        Satz-Nr. anfordern
        jsr       printline
        jsr       crlf
        lea.l     inbuff,a5        Zeige auf Tastaturpuffer
        jsr       getline          Zeile mit Nummer eintippen
        clr.l     operlh+4
        lea.l     inbuff,a5        Zeige auf Tastaturpuffer
        move.b    (a5)+,d0         1. Zeichen holen
        beq.s     numhol           erneute Eingabe!
        jsr       conv             Eingabe zu Hex machen
        cmpi.w    #maxzahl,operlh+6   Satz-Nr. zu gross?
        bcc       numhol           Bessere Eingabe holen
        jsr       crlf
        rts

* Textkonstante, zunaechst das Menue
text1   dc.b      'Befehle der Datenbank: ',cr,lf
        dc.b      'e = Eingabe in Datensaetze',cr,lf
        dc.b      'a = Anzeige eines Satzes',cr,lf
        dc.b      'ESC = Programm-Abbruch',cr,lf,0
text2   dc.b      'Bitte Satz-Nr. als Hexzahl eingeben :  ',0
text3   dc.b      'Zahl zu gross! ',0
text4   dc.b      'Name:   ',0
text5   dc.b      'Ort:    ',0
text6   dc.b      'Nummer: ',0

* Arbeitsspeicher
regsave ds.l      3                Ablage von Registern
operlh  ds.l      2                Feld des 1. Operanden
satznr  ds.w      1                Laufende Satz-Nummer
inbuff  ds.b      80               Allg. Eingabepuffer
filtflag ds.w     1                Flag-Variable
array   ds.b      banksize         Platz f. 100 Datensaetze
bankend ds.b      1                Ende der Datenbank

 INCLUDE  GEMDOSFN
 END
```

Nachwort

Die Beispiele im vorliegenden Buch mögen gezeigt haben, daß man in der Umgebung des Betriebssystems TOS leistungsfähige Lösungen für die Atari-Rechner der Serie ST entwickeln kann, wenn man in Assembler programmiert. Wesentlich ist immer, daß man die grundsätzlichen Fragen der Ein- und Ausgabe in den Griff bekommt. Dem dienen unsere "Werkzeugkästen" mit den BIOS-, XBIOS- und GEMDOS-Funktionen, für die der Vorschlag gilt, sie in geeigneter Form als Bibliothek zu speichern. Wie hier die veröffentlichten Programme zeigen, lassen sich mit ihnen bei der Ein- und Ausgabe (E/A) sowohl zeichen- als auch zeilenweise alle immer wieder benötigten Grunddienste bequem abdecken. Besonders sind hier GETLINE in "Ein Editor" und PRINTLINE im GEMDOS zu nennen, mit denen man sogar interaktiv mit Floppy-Dateien umgehen kann.

Andere hier abgedeckte Grunddienste betreffen die E/A von Zahlen und Adressen als Strings, die Zahlenwandlung und das Rechnen, das auch bei der Verwaltung von Datenfeldern (Arrays) benötigt wird. Große Bedeutung hat auch immer wieder die Bearbeitung von Textketten, für die Zeichenfilter, ein Editor, ein Parser und Suchfunktionen im Ansatz dargestellt wurden. Im Grunde stehen mit diesen Abschnitten alle Hilfsmittel bereit, um z. B. eine Datenbank oder einen Cross-Assembler aufzubauen, wenn man einmal schwierigere Projekte nennt.

Nach dem Studium der Programme und eigenen Formulierungsversuchen erkennt man, wie leistungsfähig der Befehlssatz der CPU 68000 und seine Adressierungsarten im Atari sind: Mit wenigen Programmzeilen kann viel Arbeit geleistet werden. Zur Einarbeitung braucht man aber angesichts der in Datenblättern meist etwas ungeordneten Fülle von Information die übersichtliche Gliederung und vor allem die Anleitung zum Nachvollzug. – Der Autor hofft den Lesern die Zuversicht gegeben zu haben, daß man mit dem Assembler auf den Atari-Computern gut umgehen kann. Er

erzeugt den am effektivsten ausführenden Code, und seine Ausdrucksmöglichkeiten sind stark. Die Beschreibung eines Datensatzes im Abschnitt "Array" möge ein Beispiel dafür sein, daß man für solche Zwecke nicht unbedingt die Sprache 'C' mit dem Befehl STRUCT benötigt.

Die hier dargestellte Programmierung deckt Anwendung unter TOS ab, nicht jedoch solche unter GEM, für die eine ausführliche Dokumentation hinsichtlich der notwendigen Parameterübergaben bereitstehen muß. Programmierung unter Einbindung ins GEM dürfte daher vor allem Softwarehäusern vorbehalten sein.

Zu den Assemblern sind folgende Hinweise zu geben: In jedem Falle empfiehlt sich für ihren Gebrauch ein RAM-Disk-Programm, das eine ausreichende Speichergröße bereitstellt. 256KB RAM-Disk reichen meistens nicht aus, um eine ausreichend schnelle Hantierung ohne Floppy-Zugriffe zu gewährleisten. Das Doppelte ist besser. Zweitens stand dem Autor zunächst nur der DRI-Assembler aus dem Entwicklungspaket von Atari zur Verfügung. Wegen seiner Behäbigkeit beim Assemblieren und Linken und auch wegen unerwarteter Effekte wurde er nicht mehr in Benutzung genommen, als der englische GST-Assembler samt Editor in Betrieb genommen werden konnte. Er ist auch sehr gut dokumentiert. Beide Programme sind in den Desktop unter GEM einbezogen. – Kurz vor Abschluß des Buches traf dann noch der SEKA-Assembler hier ein. Er hat enorm schnelle Durchläufe und eine sehr interaktive Bedieneroberfläche unter TOS. Er wird z. Zt. noch weiterentwickelt, um speziell auch die Direktive DS. (Reserviere Speicherplatz) einzubauen.

ANHANG

ANHANG

Definitionen für ATARI-Assembler

```
*    Definitionen fuer ATARI-Assembler
*    Nummern der Traps

GEMDOS      equ  1          BDOS-Funktionen
GEM         equ  2
BIOS        equ  13         BASIC INPUT OUTPUT SYSTEM
XBIOS       equ  14         EXTENDED BIOS

*    BIOS-Funktions-Nummern, aufzurufen mit TRAP #BIOS

GETMBP      equ  0          Memory Parameter Block (MBP) ausfuellen
BCONSTAT    equ  1          Status eines E/A-Geraetes holen
BCONIN      equ  2          Zeichen vom Eingabegeraet holen
BCONOUT     equ  3          Zeichen an Ausgabegeraet ausgeben
RWABS       equ  4          Schreiben/Lesen von Sektoren auf Diskette
SETECEC     equ  5          Setzen von Exception-Vektoren
TICKCAL     equ  6          Zeit zwischen System-Timer-Aufrufen holen
GETBPB      equ  7          Hole Zeiger auf BIOS-Parameter Block (BPB) fuer Drive
BCOSTAT     equ  8          Pruefe, ob Ausgabegeraet bereit ist
MEDIACH     equ  9          Zwischenzeitlicher Diskettenwechsel?
DRVMAP      equ  10         Pruefen, ob Drive verfuegbar (Bitmuster)
KBSHIFT     equ  11         Setzen/Pruefen von Sondertasten

*    GEMDOS-Funktions Nummern, aufzurufen mit TRAP #GEMDOS

*    Einfache lesende Funktionen
CONIN       equ  1          Lesen 1 Zch. gefiltert von Tastatur mit Echo
AUXIN       equ  3          Lesen 1 Zch. von RS232
CONIO       equ  6          Eingabe 1 Zch. mit $FF, bzw. Status holen
*                           Ausgabe 1 Zeichen mit >< $FF
DIRCON      equ  7          1 Zeichen von Tastatur roh, ohne Echo lesen
DIRCONCTRL  equ  8          Lesen wie vor mit Wirkung der CONTROL-Zeichen
READLINE    equ  10         CR-terminierten String in einen Buffer lesen
CONINSTAT   equ  11         Pruefe, ob Zeichen an der Tastatur
AUXINSTAT   equ  18         Status RS232-Eingabe pruefen

*    Einfache schreibende Funktionen
CONOUT      equ  2          Ausgabe 1 Zeichen auf Standard-Ausgabe
AUXOUT      equ  4          Ausgabe 1 Zeichen auf RS232
PRTOUT      equ  5          Ausgabe 1 Zeichen auf Centronics-Port
PRINTLINE   equ  9          0-terminierten String aus Buffer ausgeben
CONOUTSTAT  equ  16         Status der Bildschirm-Ausgabe holen
PRINTSTAT   equ  17         Status des Printers holen
AUXOUTSTAT  equ  19         Status RS232-Ausgabe pruefen

*    File-orientierte Funktionen
SETDRV      equ  14         Drive mit Nummer aktivieren
GETDRV      equ  25         Nummer aktueller Disk holen
SETDTA      equ  26         Adresse eines FILE CONTROL BLOCK setzen
GETDTA      equ  47         Adresse  des aktuellen FCB holen
DFREE       equ  54         Freien Speicherplatz der Diskette holen
MKDIR       equ  57         Ordner (Subdirectory) aus einem String bilden
RMDIR       equ  58         Entfernen eines Ordners lt. String
CHDIR       equ  59         Pfad-Namen eines Directory-Eintrages lt. String setzen
FCREATE     equ  60         Ein File mit Pfadnamen und Attribut anlegen
FOPEN       equ  61         Oeffnen eines Files
FCLOSE      equ  62         File schliessen
FREAD       equ  63         N Bytes vom File lesen
```

Fortsetzung

Definitionen für ATARI-Assembler (Fortsetzung)

```
FWRITE      equ  64      N Bytes auf File schreiben
FDELETE     equ  65      File loeschen, Name im String
LSEEK       equ  66      Zeiger in eine Datei positionieren
FATTRIB     equ  67      File-Attribut holen/aendern
DUP         equ  69      Holen des naechst freien Handles fuer Datei
FORCE       equ  70      Umlenkung der E/A
GETDIR      equ  71      Directory lt. String auf Ordner setzen

MALLOC      equ  72      Speicher reservieren/freien Speicher bestimmen
MFREE       equ  73      Speicher ab Adresse freigeben
SETBLOCK    equ  74      Reservieren eines definierten Speicherbereiches,
*                        mit Rueckgabe des Freispeichers an GEMDOS
EXEC        equ  75      (Nach-) Laden eines Programmes und ggfs. dessen Start

SFIRST      equ  78      Suchen eines Files mit Attributen
SNEXT       equ  79      Suchen eines Folgefiles mit Attributen
RENAME      equ  86      File lt. Strings umbenennen
GSDTOF      equ  87      Setzen/Holen Datum der Dateierstellung

*    Prozess beenden
TERM        equ  0       Prozess beenden, zurueck zum Caller
SUPER       equ  32      Supervisor-Modus der CPU umschalten
KEEPPROC    equ  49      Prozess abbrechen, das Programm resident lassen
PTERM       equ  76      Prozess abbrechen mit Parameteruebergabe

*    Diverse Parameter setzen/holen
GETDATE     equ  42      Datum holen
SETDATE     equ  43      Datum setzen
GETTIME     equ  44      Uhrzeit lesen
SETTIME     equ  45      Uhrzeit setzen
GETVER      equ  48      Versionsnummer Betriebssystem holen
```

Literaturhinweise

Hilf, Werner; Nausch Anton: M68000 Familie. Teil 1, Grundlagen und Architektur, te-wi Verlag, München 1984, ISBN 3-921803-16-0.

Hilf, Werner; Nausch, Anton: M68000 Familie. Teil 2, Anwendung und 68000-Bausteine, te-wi Verlag, München 1984, ISBN 3-921803-60-6.

Kane, Gerry: 68000 Mikroprozessorhandbuch. McGraw-Hill, Hamburg 1985, ISBN 3-89028-030-7.

Koch, J.: Der 16bit-Mikroprozessor SC 68000, Eigenschaften. Valvo-Buch, Hamburg 1982, ISBN 3-87095-255-5.

Koch, J.: Der 16bit-Mikroprozessor SC 68000, Befehlsvorrat. Valvo-Buch, Hamburg 1983, ISBN 3-87095-257-1.

Motorola: MC68020 32-Bit Microprocessor User's Manual. Prentice Hall, Englewood Cliffs, N. J. 1984, ISBN 0-13-541467-9.

Kane, G.; Hawkins, D.; Leventhal, L.: 68000 Assembly Language Programming. Osborne/McGraw-Hill, Berkely, C. A. 1981, ISBN 0-931988-62-4.

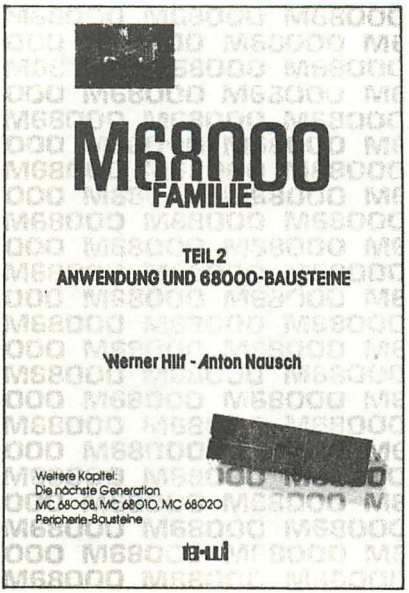
Brückmann; Englisch; Gerrits: Atari ST Intern. Data Becker, Düsseldorf 1985, ISBN 3-89011-119-X.

Digital Research: GEMDOS Spec Version 8. n. n.

Atari: The Hitchhikers Guide, 1985. Textfile im Atari-Entwicklungssystem.

Löhr, Roland: Atari 520ST empfängt seriell. In MICRO MAG, Nr. 46, Ahrensburg 1986

Motorola: M68000 Family Resident Structured Assembler, Reference Manual. Firmensschrift, Phoenix, Arizona 1984, Dokument M68KMASM/D8.



M68000-FAMILIE

Dieses zweibändige Werk befaßt sich grundlegend mit der M68000-Familie, einer Familie von leistungsfähigen 16-/32-Bit-Prozessoren und der zugehörigen Peripherie.

Im ersten Teil wird die Architektur des M68000 ausführlich behandelt. Anschließend daran folgt die Beschreibung der Adressierungsarten und des Befehlssatzes des M68000.

Teil 2 bringt Programmierbeispiele, und es werden die bisher bekannten und zukünftigen Peripheriebausteine zum 68000 behandelt. Ein interessantes Kapitel bildet die Beschreibung der weiteren Prozessoren der M68000-Familie bis hin zum 68020, dem echten 32-Bit-Mikroprozessor.
Zum Schluß erhält der Leser noch eine vollständige Beschreibung eines Single Board-Systems auf der Basis des 68000. Diese Beschreibung umfaßt die Schaltpläne ebenso wie ein vollständiges Monitor source listing. Dadurch hat jeder Interessierte die Möglichkeit, sein eigenes 68000-System aufzubauen.

M68000-Familie, von Werner Hilf und Anton Nausch
Teil 1 – Grundlagen und Architektur, 576 Seiten, DM 79,–
Teil 2 – Anwendung und 68000-Bausteine, 400 Seiten, DM 69,–

te-wi Verlag GmbH
Telefon 089/1292090

Theo-Prosel-Weg 1
8000 München 40

MEIN ATARI™-COMPUTER

(L. Poole, M.McNiff, S. Cook)

Dieses Buch macht die Möglichkeiten, die in Ihrem ATARI-Computer stecken, auf leicht verständliche Art transparent. In einfachen Schritten wird der Anwender mit der Bedienung der Geräte und mit der Software vertraut gemacht. Es enthält zahlreiche Tips zur Aufdeckung und Beseitigung von möglichen Fehlerquellen bei Hard- und Software. Scheinbar unlösbare Probleme werden beseitigt.
Dem fortgeschrittenen Programmierer bieten die vielen Übersichtstafeln, sowie die alphabetische Auflistung der BASIC-Befehle und Funktionen eine entscheidende Hilfestellung.

Ladenpreis: DM 59,–

COMPUTER FÜR KINDER, AUSGABE ATARI

Dieses Buch richtet sich an Kinder im Alter von 8 bis 13 Jahren, für deren Interesse an Computern keines der unzähligen Computer-Bücher geschrieben wurde.
„Computer für Kinder" ist ganz auf Kinder eingestellt und beschäftigt sich unterhaltsam und leicht verständlich mit folgenden Themen:

Wie arbeiten Computer
Wie funktioniert mein Computer
Wie programmiert man mit einfachen Flußdiagrammen
Wie kann ich BASIC leicht verstehen
Programme aufbauen mit Befehlen
Farbige Graphiken entwerfen
Erklärung von Computer-Begriffen

Sally Greenwood Larsen war Kindergärtnerin, ehe sie selbst Computern begegnete und zwischen den Welten von Kindern und Computern zu vermitteln begann.
Computer für Kinder, A4 quer, Fadenheftung, über 100 Seiten, DM 29,80

LOGO Computersprache für Kinder und Eltern

Dieses Buch beweist: **Jeder kann programmieren.** LOGO ist die Computersprache für Eltern und Kinder. Nicht umsonst wurde dieser Titel zum **„Buch des Jahres 1983"** in den USA. LOGO ist das Ergebnis der Erforschung menschlicher Intelligenz: entwickelt von einem Pädagogen und Mathematikprofessor. LOGO ist die erste Computersprache, die bewußt Strategien menschlichen Denkens dient.

Daniel Watt, ca. 400 Seiten, Softcover, A4, DM 59,–

---

te-wi Verlag GmbH
Telefon 089/1292090

Theo-Prosel-Weg 1
8000 München 40

MIKROCOMPUTER-GRUNDWISSEN

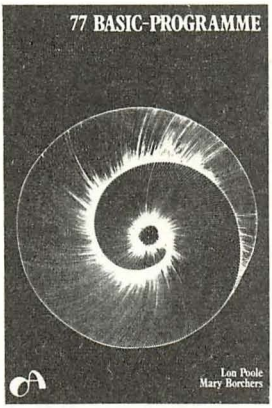
Eine allgemeinverständliche Einführung in die Mikrocomputer-Technik – vom Mikrocomputer-Papst Dr. Adam Osborne. Optimal als Einstieg für Elektronik-Laien; zur Kontaktaufnahme mit diesem die Technik und unsere Umwelt revolutionierenden Gebiet.

Von Adam Osborne
304 Seiten, DM 36,–

EINFÜHRUNG IN DIE MIKROCOMPUTER-TECHNIK

Dieses schon legendäre Standardwerk über die Mikrocomputertechnik von Erfolgsautor Osborne ist neu aufgelegt und völlig neu überarbeitet worden. Jetzt spiegelt sich darin der allerletzte Stand dieser faszinierenden Technologie wieder. In bewährter Manier ist es überaus reichhaltig bebildert. Bereits an mehr als 500 Universitäten weltweit ist es als reguläre Studiengrundlage eingeführt.

Von Adam Osborne
488 Seiten, DM 66,–



77 BASIC-PROGRAMME

Eine Sammlung von 77 praktischen Kurzprogrammen in BASIC, die mathematische, finanztechnische, statistische und verschiedene allgemeine Aufgaben behandeln. Die ausführlich erläuterten Befehle lassen sich leicht direkt anwenden oder sie dienen als Übungen.

Von Lon Poole und Mary Borchers
208 Seiten, DM 39,–

PROGRAMMIEREN IN ASSEMBLER

In diesem Buch wird die Programmierung des Mikroprozessors 6502 in Assemblersprache beschrieben. Der Titel enthält eine große Auswahl von praktischen Programmen in Standardformat einschließlich Flußdiagrammen, Quellprogrammen, Objektcodes und erläuternden Texten. Jeder Befehl der CPU wird detailliert erklärt.

Von Lance A. Leventhal
600 Seiten, DM 59,–

---

te-wi Verlag GmbH
Telefon 089/1292090

Theo-Prosel-Weg 1
8000 München 40

LOGO

Jeder kann programmieren
Computersprache für Eltern und Kinder
DANIEL WATT

LOGO ... Ergebnis der Erforschung menschlicher Intelligenz

Entwickelt von Seymour Papert, Pädagoge und Mathematikprofessor.
Erste Computersprache, die bewußt Strategien menschlichen Denkens dient – und in ihrer Logik der Realität gerecht wird. LOGO ersetzt BASIC, sagen Pädagogen und Mathematiker. LOGO kommt dem übergreifenden, assoziativen Denken entgegen. BASIC dagegen ist ein Setzkasten von Logik-Buchstaben.

DANIEL WATT ... hat im Team von Seymour Papert gearbeitet und ein Buch geschrieben, das voller Bilder seine Erlebnisse mit Kindern am Computer wiedergibt. Ein hochwertiges Textbuch für LOGO-Kurse. Ein Buch für Eltern die mit ihren Kindern nicht "Computer", sondern "Lust am eigenen Denken" erleben wollen.

"LOGO" ist auch in Ihrem ATARI.
384 S., A4-Format, DM 59,–

"Buch des Jahres 1983"
in den USA

tewi Verlag GmbH
Theo-Prosel-Weg 1
8000 München 40

SIMULATION AUF PCs

UMWELTDYNAMIK

30 Programme für kybernetische Umwelterfahrungen auf jedem BASIC-Rechner. Von Hartmut Bossel.

Umweltdynamik bezeichnet den Versuch, die Bewegungsgesetze unserer komplexen Umwelt erkennen und verstehen zu lernen. „Umwelt" meint neben Natur auch vom Menschen geschaffene Umwelten wie politische, wirtschaftliche und technische Systeme.

Dieses Buch zeigt die Kunst der Modellbildung und Simulation an Home-Computern von der sprachlichen Problembeschreibung bis zum Simulationsprogramm in BASIC! Ein BASIC-Programm, DYSYS, nimmt im Dialog die Parameter Ihres Modells entgegen und zeigt in Zeitkurven auf dem Bildschirm das Verhalten des interessierenden Systems.

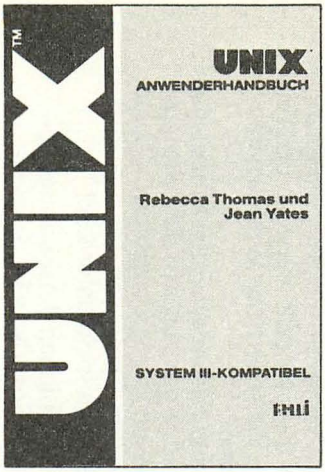
29 Modelle ökologischer Systeme werden kybernetisch erfahrbar gemacht und zeigen zugleich die Leistungsfähigkeit der Simulation auf Home-Computern, z. B. für Themen wie Baumsterben, Epidemien, Heizungsbedarf, Wirtschaftsformen usw. (Prospekt anfordern)

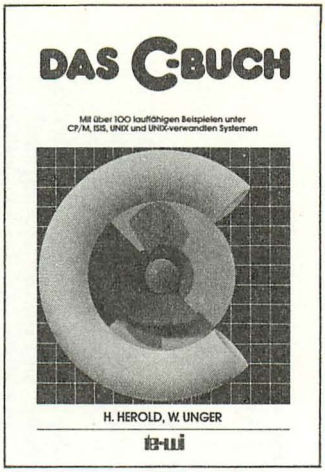
An staatlichen und kirchlichen Ausbildungseinrichtungen in Biologie, Sozialkunde, Systemanalyse eingeführter Text.

UMWELTDYNAMIK
480 Seiten, zahlreiche Abbildungen, Softcover, DM 59,–

tewi Verlag GmbH
Theo-Prosel-Weg 1
8000 München 40

Betriebssystem/Sprache der Zukunft: UNIX. . . „C"





DAS UNIX-ANWENDERHANDBUCH

der weltbekannten UNIX-Promoterin JEAN YATES zeigt die Argumente für UNIX gegenüber bisherigen Betriebssystemen und führt schrittweise vom ersten log-on zu professionellen Formen der Programmkontrolle und dem Arbeiten mit Ein-/Ausgabeprozeduren. Behandelt in pragmatischem Stil UNIX-typische Fragen wie Multitasking, Mehrbenutzersysteme, Portabilität, Modularität, Computerdatenverkehr, Utility-Programme etc.

Von Jean Yates
528 Seiten, Softcover, DM 79,–

DAS C-BUCH

Textbuch für C-Kurse und C-Anwendungen auf PCs. Beschreibt sämtliche Konstrukte der C-Sprache unter den Betriebssystemen MS DOS, CP/M, ISIS, UNIX und für die C-Compiler von MS, DR, LATTICE, INTEL. Didaktisch und typographisch außergewöhnlich. Mit über 100 lauffähigen Beispielprogrammen für PCs. Zeigt Realisierungen neuester Softwarestrategien in „C".

Von Herold/Unger
ca. 550 Seiten, Softcover, DM 79,–

---

te-wi Verlag GmbH
Telefon 089/1292090

Theo-Prosel-Weg 1
8000 München 40

DIE RECHNER DER ATARI ST-SERIE mit ihrer modernen CPU 68000 bieten eine preiswerte Möglichkeit, sich in die Welt der 16/32-Bit-Programmierung einzuarbeiten und deren Leistungsfähigkeiten einsetzen zu lernen.

DIESES BUCH vermittelt Kompetenz in der Assembler-Programmierung, einer der bedeutendsten Programmierarten, die bekanntlich den am schnellsten ausführenden Code erzeugt.
Der Text geht dabei von geringen Vorkenntnissen aus und stellt zunächst Grundelemente der Assemblersprache und der Assemblierung ausführlich vor, d.h. Befehle, Operanden, Adressierungsarten, Direktiven etc.
In der zweiten Stufe werden einfache Programme entwickelt, die der Leser zur Einübung konkret am Computer nachvollziehen kann. Sodann wird ein Werkzeugkasten mit zahlreichen, häufig benötigten Routinen eingerichtet, die die eigene Assembler-Programmierung an das Betriebssystem TOS anbinden.
Es folgen zahlreiche, sorgfältig erklärte Musterprogramme und lauffähige Dienstprogramme, die im Computeralltag benötigt werden. Behandelte Themen sind dabei u.a.: ein Editor, Fallentscheidungen, serieller Datenempfang, Arbeiten mit Floppy-Dateien, die Rechenarten u.v.a.m.

DER AUTOR, ROLAND LÖHR, Jahrgang 1930, gibt seit 1978 die Fachzeitschrift MICRO MAG heraus, die sich besonders mit der Assembler-Programmierung beschäftigt. Neben der eigenen Arbeit an zahlreichen Computertypen hat er von Anbeginn Schulungen für Mikroprozessoren und ihre Sprachen durchgeführt und seine Erfahrungen bei der übersichtlichen Darstellung des Stoffs in das vorliegende Buch eingebracht.

Weitere te-wi-Bücher

M68000 FAMILIE, 2 Bd.
Hilf/Nausch, ges. 968 Seiten
Einzige Motorola-authentische Darstellung von CPU-68000-Architektur, Programmierung, Systemaufbauten. Behandelt alle 68000-Bausteine sowie 68020, 68881.

DAS „C" BUCH (Herold/Unger)
Ein „C"-Kurs der Industrie. Für sämtliche C-Konstrukte. Über 100 Beispiele. Anspruchsvoll in Text/Bildmaterial.

LOGO – jeder kann programmieren
(Daniel Watt)
Buch des Jahres in den USA. Hochwertiges Textbuch für LOGO-Kurse für zu Hause und im Lehrbereich. Beste Rezensionen bei Kultusministerien und Lehrern.

6502 – Programmieren in Assembler
(Lance Leventhal)
Dieses Buch behandelt ausführlich die Assemblersprachen-Programmierung für den weitverbreiteten Mikroprozessor 6502. Standardwerk 7045.

UNIX (Yates/Thomas) US-Standardwerk der UNIX-Promoterin Yates. Eine sachkundige Übersicht und Einführung in die Anwendung.

UMWELTDYNAMIK
(Hartmut Bossel)
30 Programme für kybernetische Umwelterfahrungen auf jedem BASIC-Rechner. Kommentierte Modelle zu Epidemien, Waldsterben, Nahrungsketten, Wirtschaftsformen usw. Diskette für 520 ST anfordern.

Mein ATARI Computer
(Poole/McNiff/Cook)
Best-rezensiertes Standardwerk deutscher ATARI-User-Groups. Kompakte ATARI 400-/800-System/Peripheriebeschreibung.

Sprühende Ideen mit ATARI-GRAPHIK
(Tom Rowley)
Fröhlicher Lehrstoff in Geometrie und Farbenlehre eines amerikanischen Lehrers mit ATARI Graphikmöglichkeiten.

ISBN 3-921803-70-5